O'REILLY®



图灵程序设计丛书

ASP.NET Web API设计

Designing Evolvable Web APIs with ASP.NET

[美] Glenn Block Pablo Cibraro Pedro Félix Howard Dierking Darrel Miller 著

金迎 译

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

译者介绍



金迎

1997年毕业于北京大学计算机系，从事软件开发工作数年。2004年毕业于中科院计算所计算机应用技术专业，之后进入软件测试行业，具有丰富的手工和自动化测试的项目经验。


图灵程序设计丛书

ASP. NET Web API设计

Designing Evolvable Web APIs with ASP.NET

[美] Glenn Block　Pablo Cibraro　Pedro Félix
Howard Dierking　Darrel Miller 著
金迎 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（CIP）数据

ASP.NET Web API设计 / (美) 布洛克 (Block,G.) 等著 ; 金迎译. -- 北京 : 人民邮电出版社, 2015.1
（图灵程序设计丛书）
ISBN 978-7-115-37772-2

Ⅰ. ①A… Ⅱ. ①布… ②金… Ⅲ. ①网页制作工具—程序设计 Ⅳ. ①TP393.092

中国版本图书馆CIP数据核字(2014)第277931号

内 容 提 要

本书依托 ASP.NET Web API 阐述 API 设计与开发的通用技术，是一本全面介绍如何构建真实可演化 API 的实践指南。本书共分三部分。第一部分介绍 Web/HTTP 和 API 开发的基础知识，介绍 ASP.NET Web API，为初学者以及想充分利用 HTTP 的读者建立好的起点。第二部分完整介绍了真实 Web 应用程序的开发，其内容从设计讲到实现，全面覆盖客户端与服务器端开发。第三部分深入 ASP.NET Web API 的内部机制，并讲解一些高级的主题（如安全和可测试性），加深你的理解，让读者学会更好地利用 Web API 构建可演化系统。

本书适合使用 .NET、Java、Ruby、PHP、Node 等各平台 API 的开发人员学习参考。

◆ 著　　[美] Glenn Block　Pablo Cibraro　Pedro Félix　Howard Dierking　Darrel Miller
译　　金 迎
责任编辑　李松峰　毛倩倩
执行编辑　江 玥
责任印制　杨林杰
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　印刷
◆ 开本：800×1000　1/16
印张：28.5
字数：669千字　2015年1月第1版
印数：1－3 000册　2015年1月北京第1次印刷
著作权合同登记号　图字：01-2014-7512号

定价：99.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第二部分 真实世界的 API 开发

作者简介

Glenn Block 曾在 ASP.NET 团队工作，负责开发 ASP.NET Web API 的早期版本，现在就职于 Splunk，改善开发人员访问大数据的能力。作为一名拥有近 20 年经验的职业程序员，Block 一直在努力改进开发人员的工作。Block 醉心编码，据说“极少睡觉”，还是云开发的积极支持者，在微软 Windows Azure 对 OSS 栈的支持中起了关键作用。Block 是 Node.js 和 .NET OSS 项目的热心参与者、社区支持者，还经常在国际会议上发言。Block 与妻女现居住在西雅图。

Pablo Cibraro 是 Tellago 公司的一位软件架构师，也是一位享有国际声誉的专家，拥有十余年系统架构和使用微软技术实施大型分布式系统的经验。Cibraro 在过去几年间与微软公司的模式和实践团队直接合作，为使用 Web 服务、Web 服务增强（WSE）和 WCF 构建面向服务的应用程序提供示例程序、模式和指南。

Pedro Félix 居住在葡萄牙里斯本，教授和研究编程与计算机安全相关的课题。

Howard Dierking 是微软 WCF Web API 团队的产品经理，主要负责 AppFabric。在此之前，Dierking 在微软担任过其他多个角色，其中有《MSDN 杂志》的主编和 Microsoft Learning 的产品规划专员。

Darrel Miller 是 Tavis 软件公司的合伙人，这是一家专为制造业提供软件解决方案和服务的公司。Miller 的工作是向人们展示如何在业务应用程序中采用 REST 架构风格。

关于封面图

本书封面上的动物是疣螈（学名 Triturus cristatus），又名北螈或巨冠蝾螈。这种两栖动物分布于欧洲北部，从英国一直到黑海。疣螈是生活在不列颠群岛上的三种蝾螈中最大最稀有的一种，在当地受到生物多样性行动计划的保护。生物多样性行动计划致力于将濒危物种进行统计并形成保护计划。

疣螈大部分时间呆在陆地上，但是会回到池塘中进行繁殖。幼虫大约三周后孵化，并在水下生长一段时间，在四个月大时经历蜕变，成为呼吸空气的青年疣螈，离开池塘上岸居住。在陆地上，疣螈捕食昆虫及其幼虫。成年疣螈还会在池塘中捕食其他蝾螈、蝌蚪、幼年青蛙、昆虫或田螺。

因其防御能力相对较弱，疣螈喜欢居住在有植被覆盖的陆地上，例如灌木丛、草丛和茂密的丛林。雌性疣螈体积比雄性大，长度可以达到 15 厘米。雄性和雌性疣螈都具有相同的颜色图案：背部和侧面为深灰到黑色，腹部为黄色或橘红色，并有黑色斑点。在繁殖期，雄性疣螈背部生出锯齿状颈脊，与雌性疣螈外观不同。

这些疣螈从 10 月到 3 月冬眠，沉睡于繁殖地池塘底部淤泥中的木头和石头下面。通常，疣螈每年都返回同一个繁殖地，一般不会离开出生地半英里的范围。虽然有些疣螈可以活 30 年之久，但是在野外大部分只能存活约 10 年。

序

1989 年 3 月，当 Tim Berners-Lee 在欧洲核子研究组织（CERN，http://info.cern.ch/）首次提出 Web 的概念（http://www.w3.org/History/1989/proposal.html）时，就引发了一场创造性和机遇的社交革命。这场革命从此横扫全球，改变了社会的运作方式和人们的交互方式，也改变了我们如何看待自己作为个体在社会中承担的角色。

但是，Berners-Lee 还引发了一场影响同样深远的技术革命，改变了工程师们为 Web 设计思考和构建软硬件系统的方式。Web 服务器的概念从单个计算机变成了全球云端基础设施中一个完全虚拟的部分。在云端，我们可以在任何需要的地方进行计算。同样，Web 客户端也从传统的安装了浏览器的桌面个人电脑，变成了无数可以感知物理世界并与之交互的设备，这些设备通过云端的 Web 服务器与其他设备相连。

如果我们想想 Web 经历的变化，激动人心的不仅仅是这些变化发生的令人目眩的速度，还在于这些变化的发生没有任何集中的控制或协调。也就是说，Web 经历的是演化。为了适应新的需求，新的想法和解决方案不断涌现，与旧想法进行竞争。有时候，新想法胜出，存活下来；有时候，新想法落败，自行消亡。

演化天生就是 Web 不可或缺的一部分。和大自然的演化一样，如果 Web 中的某些组件适应变化的能力较强，就更有机会保持活力，不断发展。

不仅 Web 服务器和客户端发生了变化，服务器和客户端之间的交互方式也在发生巨大的改变。过去，Web 服务器提供 HTML，由客户端展示为 Web 页面；类似地，Web 客户端将 HTML 表单提交给服务器处理，无论是处理比萨饼订单，插入一篇博客文章，还是更新缺陷管理系统中的一个问题。

这种交互模型主要使用 `GET` 和 `POST` 方法，实际上只用到了 HTTP 的一部分功能。但是，HTTP 从一开始就定义了一个范围更广的应用程序模型，全面支持与数据的交互和对数据的操控。例如：除了经典的 `GET` 和 `POST` 方法，HTTP 还定义了 `PUT`、`DELETE` 和 `PATCH` 方法，可

以编程操控资源，并与资源进行交互。

这就是我们用到 Web API 的地方：利用 Web API，Web 服务器可以提供完整的 HTTP 应用程序模型，支持对资源的编程访问，使客户端能够在多种不同情况下，以一致的方式与数据进行交互，并对其进行操控。

推动人们转向 Web API 的关键因素有两个：HTML5 和移动应用。HTML5 和移动应用都利用客户端平台的计算能力，提供引人入胜的流畅体验，同时通过后台 Web API 获取和操控数据。概括地说，过去 Web 服务器只提供静态 HTML，而现在也提供 Web API，因此客户端可以使用 HTTP 应用程序模型的全部功能，进行程序化交互。这本书要解决的问题，就是如何构建出这样的 Web API。简言之，任何人要构建针对 HTML5 应用程序以及移动应用的 Web API，都应该读一读本书。本书不仅很好地介绍了 Web API，还提供很多实用指南，指导大家使用 ASP.NET Web API 构建 Web API。此外，本书还详尽地介绍了 ASP.NET Web API 的工作原理。你也可以以本书为参考，了解如何通过 HTTP 消息处理程序、格式化程序等，对 ASP.NET Web API 进行扩展。

本书不仅包括代码展示和框架说明，还介绍了一些强大的技术，例如：TDD（Test-Driven Development，测试驱动开发）和 BDD（Behavior-Driven Development，行为驱动开发），帮助你编写应用程序，并测试、验证程序功能是否符合预期。

更棒的是，本书不仅提供了关于如何构建 Web API 的"当前"指南，还引导你逐步了解如何设计一个随着需求和约束变化演化的 Web API。解决可演化性问题的想法渗透到了 Web 运作方式的核心。

在这种环境下构建一个有效运行的 Web API 并非易事。但有一点很明确，你必须从一开始就接受一个观点：任何 Web API 都必将发生改变，而且没有人能在某一时刻控制环境中的所有因素。也就是说，如果你设计了系统的一个新版本，就把旧版本抛弃，那么必定会失去已有用户或者导致问题——你必须逐步对系统进行改进，既要继续支持旧客户端，同时又要向新客户端提供新功能。

但是，构建一个灵活的可演化软件仍然是个难题。本书极佳地阐述了如何构建可以随着需求变更和演化的现代 Web 应用程序，其介绍方式是将 Web API 与超媒体结合，这是 Web 应用程序的一个激动人心的新方向。

超媒体既是一个新概念，也是一个旧概念。我们都惯于浏览 Web 页面，寻找信息，然后点击一个链接，打开新的页面，获得更多的信息和链接，深入了解感兴趣的方面。随着信息发生改变或演化，Web 页面可以加入新链接，或者修改已有链接以反映这些变化。这些新链接可以为你提供新信息，深入更多的领域。

将 Web API 与超媒体相结合，你将得到一个强大的模型，应用程序可以像 Web 页面一样变

化，适应和调整程序与服务器交互的方式。现在，客户端不再有固定的操作流程，而是根据可用的链接修改自己的行为，以进行演化；简言之，客户端能够适应变化。

对此，本书提供了一个全面的概览，介绍设计可适应双方（提供商和消费者）需求变化的 Web API 的最先进方法。通过介绍相关概念，例如使用测试驱动开发构建超媒体驱动的 Web API，本书为需要构建 Web API 的开发者提供了一个很好的起点。

作为 ASP.NET Web API 开发团队的一员，我很高兴与本书的作者一起工作。本书的作者非常优秀，他们不但都具有构建框架的经验，而且拥有构建基于 HTTP 概念的实际系统的丰富实战经验。本书的作者提出了很多极有价值的意见和建议，帮助 ASP.NET Web API 成为构建现代 Web 应用程序的流行框架。

我与 Glenn Block 的合作特别愉快，Block 很早就加入了 ASP.NET Web API 项目，极大推动了项目对社区参与的重视，以及对依赖注入、测试驱动开发和超媒体重要性的认识。没有 Block 的贡献，ASP.NET Web API 就不会达到今天的高度。

如果你正在或者考虑构建 Web API，那么本书不仅可以作为一个学习工具，而且可以用作实际的指南，帮助你基于 ASP.NET Web API 构建现代 Web 应用程序。本书提供了大量的信息和指导原则，将教你以一种新颖的方式看待复杂问题，在设计中时刻考虑可演化性。我自己非常期待看到本书在未来如何演化！

——Henrick Frystyk Nielsen

前言

为什么要阅读本书

Web API 开发呈现爆炸式增长。各家公司都在投资构建可以通过 Web 使用多种客户端访问的系统。想想你经常光顾的网站，这些网站很可能已经提供了访问 API。创建一个使用 HTTP 进行通信的 API 非常简单，而挑战在第一个版本部署后才会出现。实际上，HTTP 协议的制定者对这个问题，以及如何设计可演化 API 已经进行了周详的考虑。媒体类型和超媒体就是设计可演化 API 的核心。但是，很多 API 开发者并未对此加以考虑或利用，部署的 API 没有合理地使用 HTTP 协议，客户端严重依赖 API 的具体实现。这样的 API 很难进行演化，极易破坏客户端功能。为什么会出现这样的情况呢？因为人们经常觉得，从工程角度看，这是实现功能的最简单、最直接的做法。但是，从长期看，这种做法违反直觉，与 Web 自身的基本设计原则背道而驰。

本书的目标读者是希望设计出适应长期变化的 API 的开发者。变化是不可避免的：你今天构建的 API 将会演化。因此，问题不是“要不要”，而是“如何”设计可演化的 API。你在项目早期做出的决定（或者未做出的决定）会极大地影响以下问题的答案。

- 添加一个新功能，会破坏现有的客户端，强制现有客户端升级并重新部署吗？或者，现有客户端还能继续工作？
- 如何保障 API 的安全？能够使用较新的安全协议吗？
- API 可以扩展规模，满足用户需求吗？或者必须重新进行架构设计？
- 能够支持未来出现的新客户端和新设备吗？

你在设计时可以考虑这些问题。初看起来，这似乎是预先做大量设计（Big Design Up Front）或者瀑布方法。其实不然。你不需要在构建系统前做出完备的设计，也不需要进行过度分析。有些决定的确需要预先做出，但这些决定处于较高的层次，关系到整体设计。要做出这些决定，你并不需要理解或预知系统的每个方面。实际上，这些决定奠定了迭代演化的

基础，在随后构建系统时，你可以在此基础上，采取各种方式不断强化自己的目标。

本书更偏重应用而非理论。我们希望你读完本书之后，获得构建真实可演化系统的能力。为了达到这个目的，开篇将介绍 Web 和 Web API 开发的一些必知内容，然后从设计到实现，逐步介绍如何使用 ASP.NET Web API 创建一个新 API。这个 API 的实现将覆盖一些重要的主题，例如：如何使用 ASP.NET Web API 实现超媒体、如何执行内容协商。我们将演示这个 API 在部署后如何实际进行演化，还将讨论如何使用既有实践（例如：验收测试、测试驱动开发）和技术（例如：控制反转）提高代码的可维护性。最后，我们将深入 Web API 的内部，帮助你加深理解，更好地利用 Web API 构建可演化系统。

预备知识

要充分理解本书的内容，你应该是一位开发人员，拥有使用 .NET 3.5 或更高版本开发 C# 应用程序的经验。如果你还具有构建 Web API 的经验，那就更好了。在开发 API 时使用过哪种框架并不重要，重要的是应该熟悉相关的概念。阅读本书并不需要 ASP.NET Web API 或 ASP.NET 经验，但熟悉 ASP.NET MVC 的确会对你很有帮助。

如果你不是一位 .NET 开发人员，那么本书也有适合阅读的内容。我们编写本书时设定了一个具体的目标，要使书中大部分内容关注 API 设计和开发的通用技术，不与 ASP.NET Web API 直接相关。因此，我们认为，无论你的开发背景是什么（Java、Ruby、PHP、Node 等），都可以通过本书前两部分的大部分内容学习 API 的开发。

漫游指南

在开始你的阅读旅程之前，这里有一些漫游本书内容的指南。

- 第一部分主要对 Web API 开发进行介绍。这部分覆盖了 Web/HTTP 和 API 开发的基础知识，介绍 ASP.NET Web API。如果你刚刚接触 Web API 开发 /ASP.NET Web API,那么这部分是一个很好的开始。如果你已经在使用 ASP.NET Web API(或其他 Web API 框架)，但是想更多地了解如何充分利用 HTTP，这部分也是一个很好的起点。
- 第二部分关注真实世界的 Web API 开发。这部分完整介绍了一个真实世界中 Web 应用程序的开发，从设计到实现，覆盖客户端和服务器。如果你已经对 Web API 开发颇为熟悉，希望很快开始构建应用程序，那么可以直接从第二部分开始阅读。
- 第三部分是一个相当全面的参考资料，详细介绍了 ASP.NET Web API 各部分的内部机制。这部分还覆盖了一些较为高级的主题，例如：安全和可测试性。如果你已经在使用 ASP.NET Web API 构建应用程序，希望了解如何将 Web API 自身的功能发挥到极致，请从第三部分开始阅读。

本书内容

本书分为三部分。

第一部分　基础知识

- 第 1 章　因特网、万维网和 HTTP 协议

  这一章开篇简单回顾了万维网和 HTTP 协议的历史，然后概要介绍了 HTTP 协议。你可以把这一章看成 HTTP“傻瓜指南”。这一章提供了你需要了解的关键知识，以免去你阅读整个 HTTP 规范的辛苦。

- 第 2 章　Web API

  这一章首先大致介绍了 Web API 开发的历史背景，然后讨论 API 开发的精髓，从核心概念开始，一直深入探讨到设计 API 的风格和方法。

- 第 3 章　ASP.NET Web API 基础

  这一章讨论了 ASP.NET Web API 框架背后的基本驱动因素，然后介绍 ASP.NET Web API 的基础知识，以及 .NET HTTP 编程模型和客户端。

- 第 4 章　处理架构

  这一章将简要介绍一个 HTTP 请求在 ASP.NET Web API 中进行处理的生命周期。你将了解到处理 HTTP 请求和响应的不同方面涉及的各个不同部分。

第二部分　真实世界的API开发

- 第 5 章　应用程序 + 第 6 章 媒体类型选择与设计

  这两章讨论了问题跟踪应用程序的整体设计，涵盖了设计相关的几个重要主题，其中有：媒体类型选择和设计，以及超媒体。

- 第 7 章　构建 API + 第 8 章 改进 API

  这两章展示了如何使用 ASP.NET Web API 实现和改进超媒体驱动的问题跟踪 API，并介绍了如何使用行为驱动的开发方式。

- 第 9 章　构建客户端

  这一章介绍如何构建问题跟踪 API 的一个超媒体客户端。

第三部分　Web API细节

- 第 10 章　HTTP 编程模型

  这一章将深入讨论 ASP.NET Web API 依托的 .NET HTTP 编程新模型。

- 第 11 章　托管

  这一章介绍了 ASP.NET Web API 适用的所有托管模型，其中有：自托管、IIS 以及新的 OWIN 模型。

- 第 12 章　控制器和路由

  这一章深入探讨了 Web API 路由的工作机制，以及控制器如何运作。

- 第 13 章　格式化程序和模型绑定 / 第 14 章 HttpClient

  这两章介绍了关于模型绑定和使用新的 HTTP 客户端类的所有需知内容。

- 第 15 章　安全 / 第 16 章 OAuth 2.0 授权框架

  这两章介绍了 ASP.NET Web API 的整体安全模型，然后详细讨论了如何在 API 中实现 OAuth 规范。

- 第 17 章　可测试性

  这一章将介绍如何以测试驱动的方式，在 ASP.NET Web API 中进行开发。

排版约定

本书使用了下述排版约定。

- 楷体

  标示新术语。

- 等宽字体（`constant width`）

  表示程序片段，也用于在正文中表示程序中使用的变量、函数名、命令行代码、环境变量、语句和关键字等元素。

- 等宽粗体（**`constant width bold`**）

  表示应该由用户逐字输入的命令或者其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文决定的值替换的文本。



这个图标代表提示或建议。



这个图标代表重要说明。



这个图标代表警告或提醒。

使用代码示例

读者可以在这里下载本书随附的资料（代码示例、练习题等）：https://github.com/webapibook。讨论本书的论坛地址为：https://groups.google.com/forum/#!forum/webapibook。

希望本书能够助你一臂之力。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则你不必与我们联系取得授权。例如，无需请求许可，你就可以用本书中的几段代码写成一个程序。但是销售或者发布 O'Reilly 图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需获得授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处，但不强求。出处一般包括书名、作者、出版商和 ISBN，例如：*Designing Evolvable Web APIs with ASP.NET* by Glenn Block, Pablo Cibraro, Pedro Felix, Howard Dierking, Darrel Miller (O'Reilly). Copyright 2012 Glenn Block, Pablo Cibraro, Pedro Felix, Howard Dierking, and Darrel Miller, 978-1-449-33771-1。

如果还有关于使用代码的未尽事宜，你可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品（http://www.safaribooksonline.com/content）。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体（http://www.safaribooksonline.com/organizations-teams）、政府机构（http://www.safaribooksonline.com/government）和个人（http://www.safaribooksonline.com/individuals），Safari Books Online 提供各种产品组合（http://www.safaribooksonline.com/subscriptions）和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社（http://www.safaribooksonline.com/publishers）的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见（http://www.safaribooksonline.com/）。

联系我们

请把对本书的意见和疑问发送给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/designing-api

如果你对本书有一些建议或技术上的疑问，请发送电子邮件至 bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly
请关注我们的 Twitter 动态：http://twitter.com/oreillymedia
我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

完成这本书，实际的工作量远超我们的预计。首先要感谢我们的妻子和孩子，他们给予了

我们大量的耐心，以及写作的时间和空间。

如果没有大家的审阅和指导，我们不可能完成这本书，在此要感谢：Mike Amundsen、Grant Archibald、Dominick Baier、Alan Dean、Matt Kerr、Caitie McCaffrey、Henrik Frystyk Nielsen、Eugenio Pace、Amy Palamountain、Adam Ralph、Leonard Richardson、Ryan Riley、Kelly Sommers、Filip Wojcieszyn 和 Matias Woloski。

第一部分

基础知识





第 1 章

因特网、万维网和HTTP协议

要掌握 Web，就要理解其基础和设计。

让我们从源头开始了解 Web API。在 20 世纪 60 年代后期，DARPA（Defense Advanced Research Projects Agency，美国国防部高级研究计划局）创建了以 TCP/IP 协议连接的一组基于网络的系统，即 ARPANET（Advanced Research Projects Agency Network，http://www.cs.utexas.edu/users/chris/think/ARPANET/）。ARPANET 的设计初衷是在美国的大学和实验室中共享数据（参见图 1-1）。

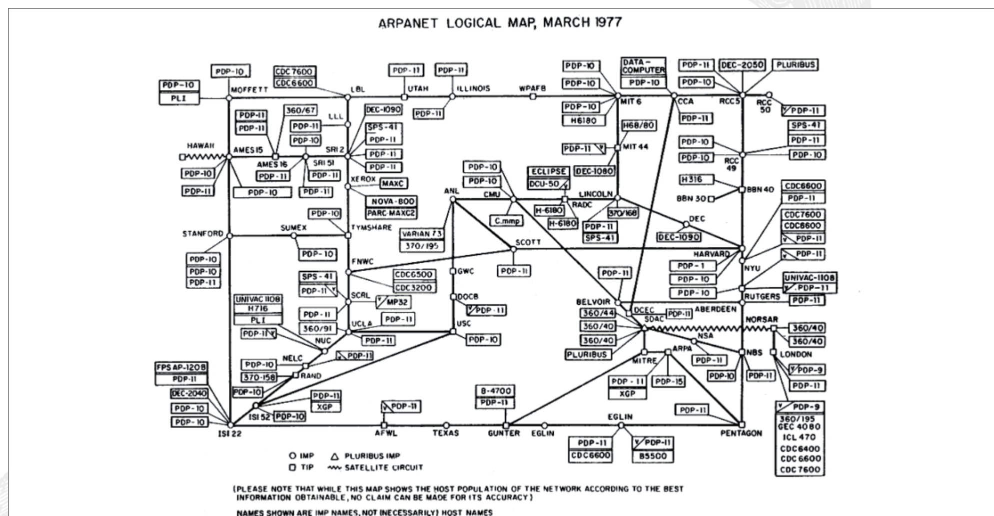


图 1-1：ARPANET（图片来自维基共享资源）

ARPANET 不断完善，终于在 1982 年衍生出一个全球性的互联网络，即因特网。因特网建立在一组通信协议——TCP/IP（Internet protocol suite，因特网协议族）的基础上。虽然 ARPANET 是一个相当封闭的系统，但因特网被设计为一个全球性的开放系统，以连接公私单位、组织、机构及个人。

在 1989 年，欧洲核子研究组织（CERN）的科学家 Tim Berners-Lee 创建了一个叫 Web（World Wide Web，万维网）的新系统，能够通过 Web 浏览器访问因特网上链接的文档。由于 Web 文档大多用 HTML 语言书写，浏览这些文档需要一个特殊的应用协议：HTTP（HyperText Transfer Protocol，超文本传输协议）。这种协议是网站运行和 Web API 工作的核心。

本章，我们将深入了解 Web 体系结构的基本知识，探究 HTTP 协议。这将为我们下一步设计 Web API 奠定基础。

1.1 Web体系结构

如图 1-2 所示，Web 有三个核心概念：资源（resource）、URI（Uniform Resource Identifier，统一资源标识符）和表示（representation），参见 http://www.w3.org/TR/webarch。

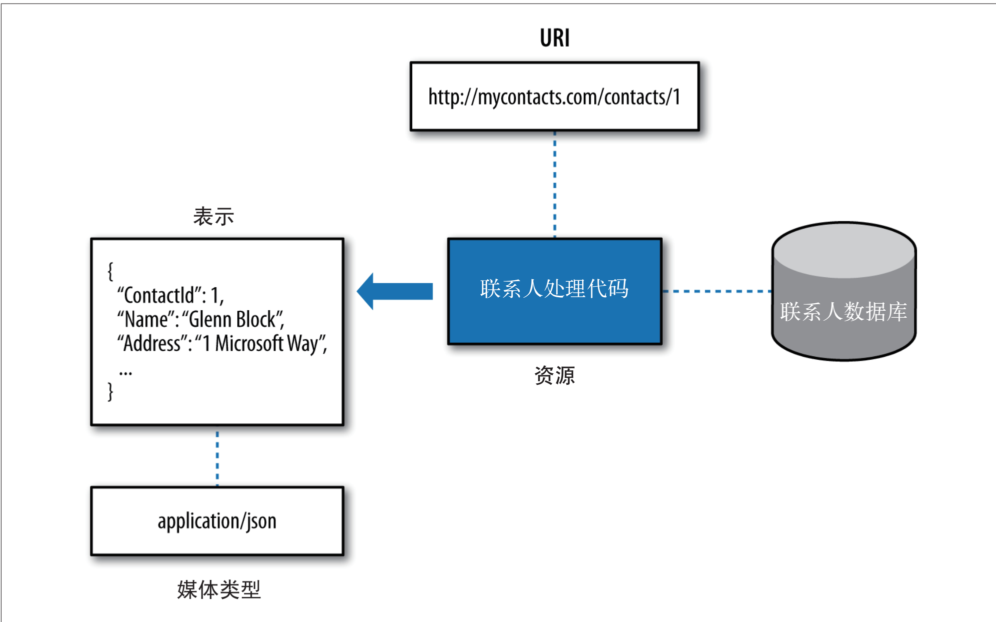


图 1-2：Web 核心概念

一个资源由一个 URI 进行标识，而 HTTP 客户端使用 URI 就可定位资源。表示是从资源

返回的那些数据。和 Web 相关的另一个重要概念是媒体类型（media type），指的是从资源返回数据的格式。

1.1.1 资源

任何带有 URI 标识的东西都是资源。资源自身是一个或多个实体的概念性映射（http://tools.ietf.org/html/rfc2396#section-1.1）。早年的 Web 中，资源映射的实体通常是一个文件，如一份文档或者一个网页。不过，资源并不只限于文件。它可以是一个可建立连接的服务，通过它可访问产品目录、连接设备（如打印机）或者车库门无线遥控器，也可以是一个像 CRM（Customer Relationship Management，客户关系管理）或采购系统那样的内部系统。资源还可以是一个流媒体，如视频流或者音频流。

资源必须关联实体或者数据库吗？

现在，人们对于 Web API 有一个常见的误解，认为一个资源必须对应到一个有数据库支持的实体或者业务对象。在讨论设计时经常会有人说："我们不能用这个资源，因为这个资源需要在数据库中创建一个表，而我们又不需要这张表。"前面给出的资源定义中描述了到一个或多个实体的映射，这里所说的实体是泛指的（也就是说，实体可以是任何东西），而不是特指业务对象。一个应用程序可以这样设计，让其中提供的资源总是映射到业务实体或数据库表，对这种系统而言，前面那种说法确实是正确的。但是，这种限制是由应用或框架强加的，Web API 本身并没有这种限制。

在构建 Web API 时，这种实体 / 资源的限制在很多情况下会导致问题。例如，一个订单处理资源，在处理一个订单时实际上会统筹实施不同的系统。在这种情况下，执行这个资源会调用系统的几个部分，这些部分会各自在数据库中存储状态，而这个资源自身可能在数据库中存储状态，也可能不存。问题的关键是，这个资源在数据库中没有直接对应关系。并且，被调用的各个组件也并不一定要使用数据库（虽然在前面例子中是使用了数据库的）。

在设计 Web API 时，请记住上述区别，这可以帮助你在系统中真正发挥 Web 的力量。

1.1.2 URI

如前所述，每个资源都可以通过唯一的 URI（http://tools.ietf.org/html/rfc3986）访问。你可以把 URI 看成一个资源的主键（primary key）。URI 的例子有：http://fabrikam.com/orders/100、http://ftp.fabrikam.com、mailto:John.Doe@example.com、telnet://192.168.1.100 以及 urn:isbn:978-1-449-33771-1。一个 URI 只能对应一个资源，但是多个 URI 可以指向同一个资源。每个 URI 的格式都是：*scheme:hierarchical part[?query][#fragment]*，其中查询字符串"query"和"fragment"是可选的，这里的"Hierarchical part"还可以包含安全证

书颁发机构（authority）和分层路径（hierarchical path）。

URI 分为两种类型：统一资源定位符和统一资源名。URL（Universal Resource Locator，统一资源定位符）既标识一个资源，又指定了访问该资源的方法，而 URN（Universal Resource Name，统一资源名）仅仅是一个资源的唯一标识符。前面给出的 URI 例子中，最后一个例子是本书的 URN，其他都是 URL。这个 URN 只标识了本书，但是不包含关于如何获取该书的信息。在实际应用中，你看到的大多数 URI 都是 URL，因此 URI 和 URL 常常同义替换使用。

是否使用查询字符串？

有一个经常被讨论的问题：要不要在 URI 中使用查询字符串？这个问题的回答和缓存机制有关。有些缓存会自动忽略任何带有查询字符串的 URI。也就是说，所有带查询字符串的 URI 请求都会转到源服务器（origin server）进行处理，而这会对系统的扩展产生重要影响。于是，有些人倾向于不使用查询字符串，而把这部分信息放在 URI 路径中。谷歌推荐的做法（参见 https://developers.google.com/speed/docs/insights/LeverageBrowserCaching#LeverageProxyCaching）是：对静态资源不要使用查询字符串，以便缓存。

1.1.3　酷URI

酷 URI（http://www.w3.org/TR/cooluris/#cooluris）是简单易记而且不变的 URI（例如 http://www.example.com/people/alice）。如果 URI 发生改变，那么链接到这个 URI 的现有系统将不能继续工作，所以有些 URI 需要保持不变。如果你希望客户端保存指向你的资源的书签，那么就应该考虑使用酷 URI。如果你有一些网页，常常被其他网站添加链接，或者被用户存储在浏览器收藏夹里，那么，酷 URI 会非常好用。但是，不是所有情况下都要用到酷 URI。你在这本书中将会看到，不必用很多酷 URI，API 设计也可以做得很好。

1.1.4　表示

表示是资源在某个时刻状态的快照。当 HTTP 客户端请求一个资源时，返回的是这个资源的表示，而不是资源本身。从一个请求到下一个请求发生时，资源的状态可能会发生很大的变化，因而返回的表示也会大不相同。例如，假设有一个提供开发者文章的 API，通过访问 URI *http://devarticles.com/articles/top* 得到排名第一的文章，这个 API 返回的不是指向该 URI 内容的链接，而是到那篇文章的重定向信息。随着时间的推移，文章排名发生变化，（重定向返回的）资源的表示也会随之改变。在这个例子里，资源并不是那篇文章，而是服务器上运行的逻辑，这个逻辑从数据库中获取排名第一的文章，从而返回重定向信息。需要注意的是，一个资源可以有一个或多个表示，详见 1.2.8 节。

1.1.5 媒体类型

每个表示都有特定的格式，即*媒体类型*（media type）。媒体类型是在因特网上客户端和服务器之间传递信息的格式。媒体类型由两部分标识组成，例如 `text/html`。媒体类型有多种用途。有些媒体类型非常通用，例如，`application/json`（表示一组值或一组键值）或 `text/html`（主要用于在浏览器中显示文档）。另一些媒体类型的语法限制较多，例如，`application/atom+xml` 和 `application/collection+json`，专门用于管理源和列表。还有用于 PNG 图像的 `image/png` 媒体类型。媒体类型也可以是专属于特定领域的，例如 `text/vcard` 用于名片和联络信息的电子化共享。附录 A 列出了一些常见的媒体类型。

媒体类型自身实际上包含两部分。第一部分（斜线前）是*顶级媒体类型*，这部分描述了通用的类型信息以及常用处理规则。常见的顶级类型有：`application`、`image`、`text`、`video` 和 `multipart`。第二部分是*子类型*（subtype），描述一个非常具体的数据格式。以 `image/png` 和 `image/gif` 为例，它们的顶级类型告诉客户端这是一个图像（image），而子类型 `png` 和 `gif` 具体说明了这是什么类型的图像，应该如何处理。子类型经常有不同的变种，使用一样的语法，但格式不同。例如，HAL（Hypertext Application Language，超文本应用程序语言，http://stateless.co/hal_specification.html）有两个变种：JSON（`application/hal+json`）和 XML（`application/hal+xml`）。子类型 `hal+json` 说明该 HAL 使用 JSON 传输格式，而 `hal+xml` 说明使用的是 XML 传输格式。

> **媒体类型的起源**
>
> *媒体类型最早源自 ARPANET。最初，ARPANET 是通过简单文本信息进行通信的计算机网络。系统不断扩展，开始需要更为多样的通信。于是，标准格式（http://tools.ietf.org/html/rfc822）被制定出来，使信息可以包含与其处理相关的元数据。随着时间的推移和电子邮件的产生，这个标准演变为 MIME（Multipurpose Internet Mail Extension，多用途因特网邮件扩展，http://tools.ietf.org/html/rfc2045）。MIME 的用途之一是支持非文本格式内容，从而产生了媒体类型（http://tools.ietf.org/html/rfc2046），用于描述一个 MIME 实体的主体。随着因特网的蓬勃发展，人们不必使用电子邮件就可以在 Web 上传送同样丰富的信息，因此媒体类型也开始用于描述 HTTP 请求和响应的主体，从而与 Web API 联系在了一起。*

媒体类型注册

通常媒体类型是注册在一个由 IANA（Internet Assigned Numbers Authority，因特网号码分配局）管理的中央注册库（http://www.iana.org/assignments/media-types/media-types.xhtml）中。这个注册库本身包含一份媒体类型列表，以及到相关说明书的链接。注册表按照顶级媒体类型进行分类，每个顶级分类包含一份具体媒体类型的列表。

应用程序开发者要设计能够识别标准媒体类型的客户端或服务器，就可以参考这个媒体

类型注册库。例如，如果你要创建一个识别 `image/png` 媒体类型的客户端，可以导航到 IANA 媒体类型页面（http://www.iana.org/assignments/media-types/media-types.xhtml）的 image 分类，查找“png”，得到 `image/png` 类型的说明，具体页面如图 1-3 所示。

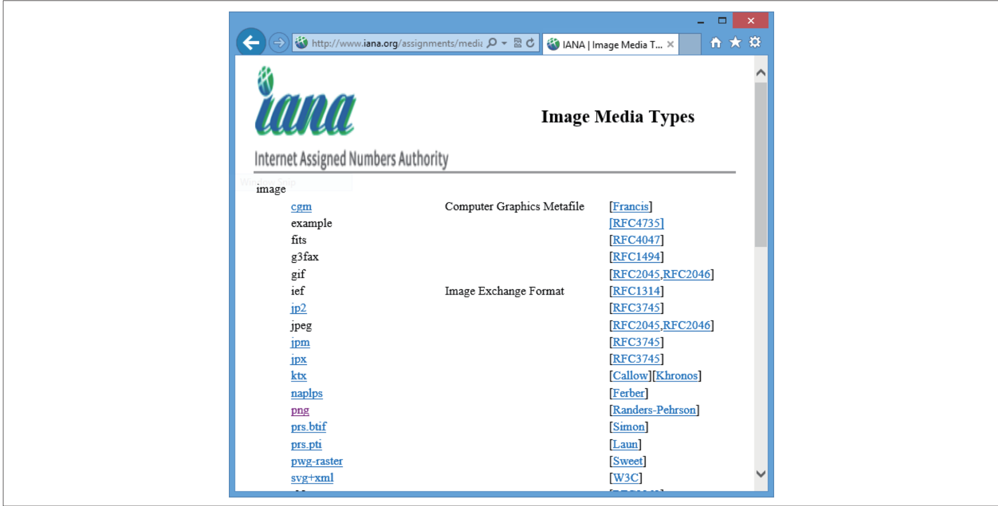


图 1-3：IANA 注册库的 image 分类

为什么要有这些不同的媒体类型呢？因为每个类型都有各自的优点，或者有其量身定做的客户端。HTML 类型展示文档（例如：Web 页面）效果极佳，但不一定最适合于传输数据。JSON 传输数据很好用，但是在重现图像上效率却十分低下。PNG 是极好的图像格式，但在存储可扩展的矢量图形方面不大理想，对此 SVG 才是优选。比起不成熟的 XML 或 JSON，ATOM、HAL 和 Collection+JSON 能表达更为丰富的应用程序语义，不过，它们受到的限制也较多。

读到这里，你已经了解了 Web 体系结构的核心组件。下一节我们将详细了解 HTTP——将所有这些组合起来的黏合剂。

1.2　HTTP协议

介绍完大体的 Web 体系结构，我们接下来看 HTTP。HTTP 协议涵盖广泛，因此我们并不试图介绍所有内容，而是关注一些主要概念，特别是那些和构建 Web API 相关的概念。如果你初识 HTTP，这部分内容应该正好可以帮你打下基础；如果不是，你也应该能习得原本不知道的知识，不过跳过这部分内容也无妨。

HTTP（http://www.w3.org/Protocols/rfc2616/rfc2616.html）是信息系统的应用层协议，是驱动 Web 的核心。HTTP 协议原本由三位计算机科学家提出，他们是：Tim Berners-Lee、Roy Fielding 和 Henrik Frystyk Nielsen。HTTP 协议定义了供客户端和服务器在网络上传输

信息的统一接口，让它们无需了解执行的具体细节。HTTP 协议是为支持动态变化的系统而设计的，可以容忍一定程度的延迟和陈旧。这种设计允许中间层（例如：代理服务器）介入通信，提供各种功能，例如缓存、压缩以及路由选择。万维网规模庞大，不断变化，而网络拓扑持续展开，延迟不可避免，而 HTTP 协议因其特性，正好成为万维网的理想协议。自 1996 年诞生时起，HTTP 协议就一直驱动着万维网的运行，经受住了时间的考验。

1.2.1 HTTP 1.1之后

HTTP 不是一成不变的，我们理解以及使用它的方式都在进化。由于文本的模糊性，某些时候甚至被认为错误，围绕 HTTP 协议规范 RFC 2616，存在着很多的误解。IETF（Internet Engineering Task Force，互联网工程任务组）成立了一个名为 httpbis（http://datatracker.ietf.org/wg/httpbis/charter/）的工作组，这个工作组创建了一套草案（http://datatracker.ietf.org/wg/httpbis/documents/），专门用于取代 RFC 2616，澄清误解。此外，这个工作组还承担着创建 HTTP 2.0 规范（http://datatracker.ietf.org/doc/draft-ietf-httpbis-http2/）的任务。HTTP 2.0 不会影响 HTTP 协议公开的表层领域，而是对底层传输进行了一组优化，例如采用新的 SPDY 协议（http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00）。因为 httpbis 的存在是为取代 HTTP 协议规范，并且，也帮助人们建立对 HTTP 更为完善的理解，我们将使用 httpbis 草案作为下文的基础。

1.2.2 HTTP消息交换

基于 HTTP 协议的系统以一种无状态的方式，使用请求 / 响应模式进行信息交换。我们将简单介绍这一交换过程。首先，一个 HTTP 客户端生成一个 HTTP 请求，如图 1-4 所示。

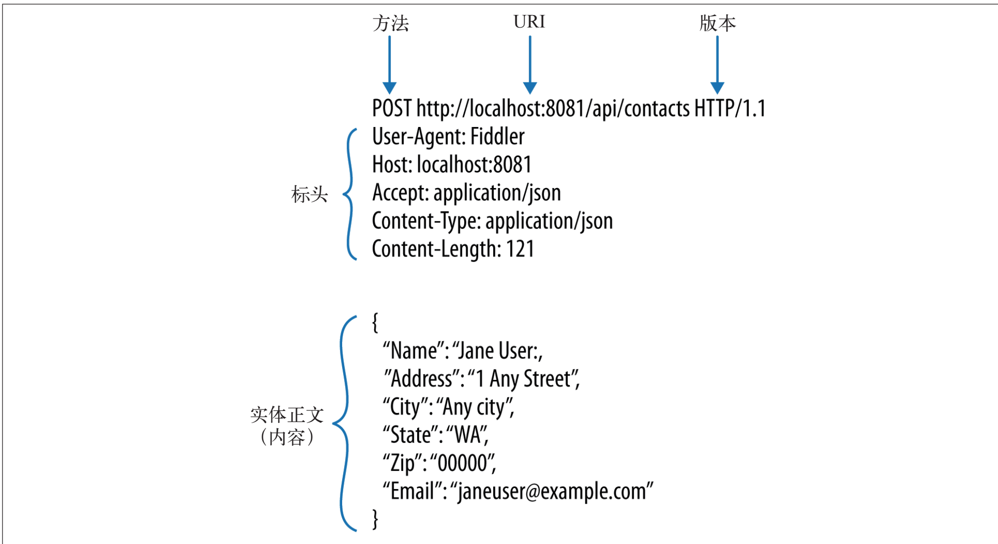


图 1-4：HTTP 请求

这个 HTTP 请求是一个消息，其中包含一个 HTTP 版本、一个所访问资源的 URI、请求标头（header）和一个 HTTP 方法（如 GET），还可以包含一个可选的实体正文（内容）。这个请求随后发送到资源所在的源服务器（origin server）。服务器查看 URI 和 HTTP 方法，以判断自己是否能够处理这个消息。如果服务器可以处理这个消息，就查看请求标头中包含的控制信息（如内容描述），然后基于这些信息处理消息。

服务器完成消息处理之后，就生成了一个 HTTP 响应，通常包含所请求资源的一个表示（如图 1-5 所示）。

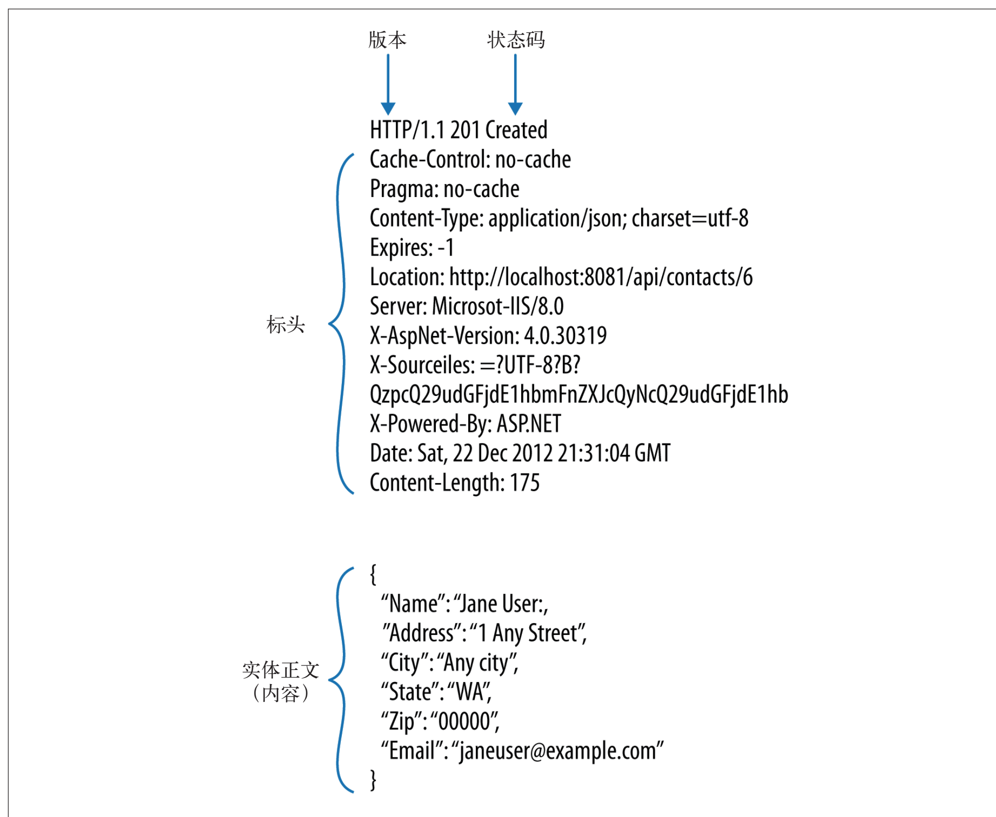


图 1-5：HTTP 响应

HTTP 响应包含 HTTP 版本、响应标头、可选的实体主体（其中包含资源表示）、一个状态码和一个描述。与收到消息的服务器类似，客户端会检查响应标头，使用其控制信息对消息及其内容进行处理。

1.2.3　中间层

前述的 HTTP 信息交换过程虽然准确，但遗漏了一个重要部分：中间层（http://tools.ietf.

org/html/draft-ietf-httpbis-p1-messaging-21#section-2.3)。HTTP 是一个具有层次的结构，系统中的每个组件 / 服务器都有各自不同的关注点，HTTP 客户端也不需要“看见”源服务器。在 HTTP 请求向源服务器传递时，它会遇到如图 1-6 所示的中间层。中间层是一些代理或组件，检查 HTTP 请求或响应，可能对其进行修改或者替换。一个中间层可以立即返回一个响应，触发某些过程（如记录详细日志），或者只是让 HTTP 请求通过。中间层具有一些优点，可以增强或者改进通信方式。例如，缓存可以通过返回来自源服务器的缓存结果来缩短响应时间。

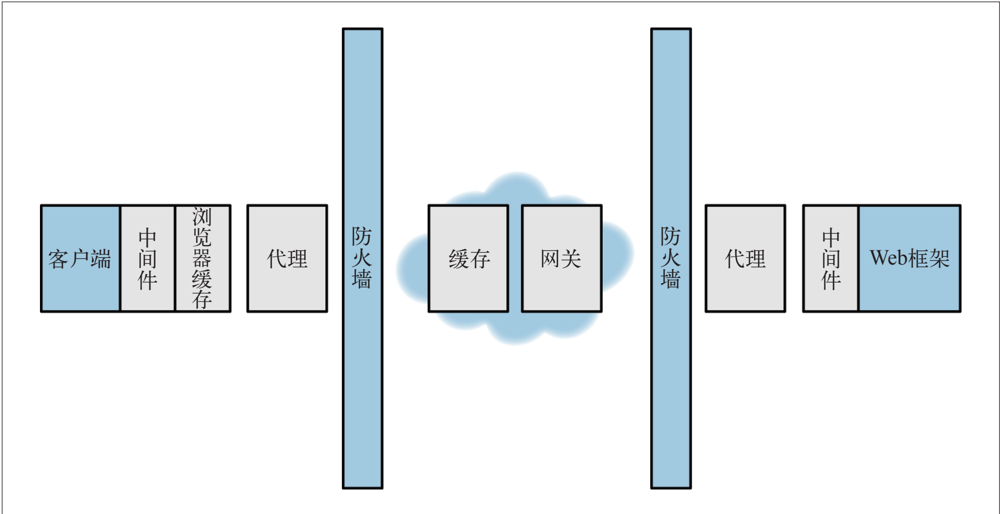


图 1-6：HTTP 中间层

请注意：中间层可以存在于 HTTP 请求从客户端到源服务器经过的任何地方，具体处在什么位置并不重要。中间层既可以和客户端或源服务器运行在同一台机器上，也可以是因特网上一台专用的公共服务器。中间层也可以内建在系统内，如 Windows 系统上的浏览器缓存，也可以是中间件插件。ASP.NET Web API 支持多种可用于客户端或服务器的中间件，如处理程序（handler）和筛选器（filter），第 4 章和第 10 章将对此进行介绍。

1.2.4　中间层类型

参与 HTTP 消息交换并对客户端可见的中间层有三种。

- 代理（proxy），它代表客户端发出 HTTP 请求并接受响应。客户端使用代理时采用主动模式，需要进行相应的配置。很多组织都会使用内部代理，组织内的用户必须通过这个代理访问因特网，这种做法非常普遍。如果一个代理有意对 HTTP 请求或者响应进行修改，那么这个代理就是*转化代理*（transforming proxy）。不修改消息的代理称为*非转化代理*（non-transforming proxy）。

- 网关（gateway）接受传入的 HTTP 消息，将其翻译为服务器底层的协议，底层协议可能是 HTTP，也可能不是 HTTP。网关也处理传出的消息，将其转化为 HTTP 协议。网关可以代表源服务器处理请求。
- 隧道（tunnel）在两个连接之间建立一个私有通道，不会修改任何消息。隧道的一个例子是两个客户端穿过防火墙用 HTTPS 进行通信。

CDN 是中间层吗？

另一种常见的因特网缓存机制是 CDN（Content Delivery Network，内容分发网络）。CDN 是一组分散的机器，缓存和返回静态内容。市场上有很多流行的 CDN 服务商，如 Akamai（http://www.akamai.com/），为公司提供内容缓存。那么，CDN 是中间层吗？答案取决于内容请求是怎样传递给 CDN 的。如果客户端直接向 CDN 发出请求，那么 CDN 扮演的是源服务器的角色。有些 CDN 功能类似于网关，对客户端是不可见的，但它实际上代表源服务器进行操作，缓存并返回所请求的内容。

1.2.5　HTTP方法

HTTP 协议提供一组标准方法（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-5.3），作为资源访问的接口。在最早的 HTTP 规范发布后，`PATCH` 方法（参见 http://tools.ietf.org/html/rfc5789）也被批准通过。如图 1-4 所示，HTTP 方法是 HTTP 请求的一部分。下面介绍 API 开发者执行的常用方法。

- `GET`

  从资源获取信息。如果返回资源，服务器应该返回状态码 `200`（`OK`）。

- `HEAD`

  与 `GET` 相同，但返回标头而非主体。

- `POST`

  请求服务器接受消息中包含的实体，交由目标资源处理。作为请求处理过程的一部分，服务器可以创建一个新的资源，但不一定如此。如果服务器创建了资源，那么应该返回状态码 `201`（`Created`）或者 `202`（`Accepted`），并返回一个地址标头，告知客户端从何处访问新资源。如果服务器没有创建资源，那么应该返回状态码 `200`（`OK`）或者 `204`（`No Content`）。在实际应用中，`POST` 方法基本上可以进行任何类型的处理，不受任何限制。

- `PUT`

  请求服务器将指定 URI 所代表的目标资源替换为消息中包含的实体。如果当前表示对应的资源存在，服务器应该返回状态码 `200`（`OK`）或者 `204`（`No Content`）。如果对应的资源不存在，那么服务器可以创建这个资源。如果服务器创建了资源，应该返回状态码

201（Created）。POST 和 PUT 的主要区别在于：POST 方法预期数据被传入加工，而 PUT 方法预期数据被替换或者存储。

- DELETE

  请求服务器移除指定 URI 所代表的实体。如果服务器立即移除了指定的资源，那么应该返回状态码 200（OK）。如果资源尚未移除，那么服务器应该返回状态码 202（Accepted）或者 204（No Conent）。

- OPTIONS

  请求服务器返回功能信息。大多数情况下，服务器返回一个 Allow 标头，具体说明支持哪些 HTTP 方法，尽管协议规范对此并没有硬性规定。例如：服务器完全可以列出支持哪些媒体类型。OPTIONS 方法也可以返回一个正文，提供在标头内无法表示的更多信息。

- PATCH

  请求服务器对指定 URI 所代表的实体进行部分更新。PATCH 方法中应该包含足够的信息，供服务器进行所请求的更新。如果指定的资源存在，服务器可以进行更新并返回状态码 200（OK）或者 204（No Content）。与 PUT 的处理方法类似，如果指定的资源不存在，服务器可以创建这个资源。如果服务器创建了资源，就应该返回状态码 201（Created）。如果一个资源支持 PATCH 方法，那么 OPTIONS 响应的 Allow 标头可以对此进行说明。服务器也可以使用 Accept-Patch 标头，列出客户端可以对其发送 PATCH 请求的媒体类型。协议规范建议媒体类型应包含语义，向服务器传递部分更新所需的信息。json-patch（参见 https://tools.ietf.org/html/draft-pbryan-json-patch-04）是一个提议的媒体类型草案，支持表达部分更新中所需的操作。

- TRACE

  请求服务器返回其收到的请求。服务器返回一个 content-type 为 message/http 的正文，其中包含完整的请求信息。客户端可以使用 TRACE 方法，查看请求消息经过的代理以及中间层对消息所做的修改，有助于进行问题诊断。

1. 条件请求

HTTP 协议的一个额外特征是可以让客户端发出附条件的请求。要进行这种请求，客户端需要发送特殊的标头，为服务器处理请求提供所需的信息。这种特殊的标头包括 If-Match、If-NoneMatch 和 If-ModifiedSince。附录 B 中的表 B-2 将详细描述这些标头。

- **条件 GET**　服务器可以通过客户端发送的标头，判断客户端缓存的资源表示是否仍然有效。如果客户端的缓存仍然有效，那么服务器不返回所请求资源的表示，而是返回状态码 304（Not Modified）。使用条件 GET 可以减少网络流量（因为响应消息很短），还可以降低服务器的负载。

- **条件** PUT　服务器可以通过客户端发送的标头，判断客户端缓存的资源表示是否仍然有效。如果客户端缓存仍然有效，服务器返回状态码 409（Preconditions Failed）。条件 PUT 可以用于并发处理。使用条件 PUT，客户端可以判断在自己发起请求时是否有另一个用户修改了数据。

2. 方法属性

HTTP 方法可以具有如下的附加属性。

- **安全**（safe）　使用安全的 HTTP 方法发送请求，从用户方面不会产生任何副作用。这并不是说安全方法根本不会产生副作用，而是说用户可以使用这个方法安全地发起请求，不用担心无意间修改了系统状态。
- **幂等**（idempotent）　通过幂等方法对资源发出一次请求，与多次请求效果相同。按照定义，所有的安全方法都是幂等的。有些方法不是安全的，但也可以是幂等的。与安全方法类似，使用幂等方法的请求并不能保证在服务器端不产生任何副作用，但是这些可能的副作用和用户无关。
- **可缓存**（cachable）　可缓存方法可以从中间层缓存处，获取对之前请求缓存的响应。

表 1-1 列出了 HTTP 方法及其是否为安全、幂等或可缓存方法。

表1-1：HTTP方法

方　法	安　全	幂　等	可缓存
GET	是	是	是
HEAD	是	是	是
POST	否	否	否
PUT	否	是	否
DELETE	否	是	否
OPTIONS	是	是	否
PATCH	否	是	否
TRACE	是	是	否

在以上列出的方法中，如今的 API 开发者最常使用的是 GET、PUT、POST、DELETE 和 HEAD。PATCH 是个新方法，但在逐渐得到普遍地使用。

设置一套 HTTP 标准方法具有如下优势。

- 只要 HTTP 资源遵循协议规范，任何 HTTP 客户端都可以与其交互。客户端可以使用方法，如 OPTIONS 获取和发现信息，从而了解如何交互。
- 服务器可以进行优化。代理服务器和 Web 服务器可以选择一些方法提供优化。例如：缓存代理知道 GET 请求可以缓存，因此，如果你发出一个 GET 请求，代理就可以返回一个缓存的资源表示，不需要把请求发送给服务器。

1.2.6 标头

HTTP 消息的标头（header）字段为客户端和服务器提供信息，用于处理这个 HTTP 请求。标头有四种类型：消息、请求、响应和表示。

- *消息标头*

  请求和响应信息都可以包含消息标头。消息标头提供的信息与消息自身而非实体正文相关，具体包括：

  - 与中间层相关的标头，如 `Cache-Control`、`Pragma` 和 `Via`；
  - 与消息相关的标头，如 `Transfer-Encoding` 和 `Trailer`；
  - 与请求相关的标头，如 `Connection`、`Upgrade` 和 `Date`。

- *请求标头*

  请求消息通常可以包含请求标头，实体正文则不包含请求标头，但 `Range` 标头除外。请求标头包括：

  - 关于请求的标头，如 `Host`、`Expect` 和 `Range`；
  - 用于身份验证信息的标头，如 `User-Agent` 和 `Form`；
  - 用于内容协商的标头，如 `Accept`、`Accept-Language` 和 `Accept-Encoding`；
  - 用于条件请求的标头，如 `If-Match`、`If-None-Match` 和 `If-Modified-Since`。

- *响应标头*

  响应消息可以包含响应标头，实体正文不包含响应标头。响应标头包括：

  - 提供目标资源信息的标头，如 `Allow` 和 `Server`；
  - 提供附加控制数据的标头，如 `Age` 和 `Location`；
  - 与所选表示相关的标头，如 `ETag`、`Last-Modified` 和 `Vary`；
  - 与认证挑战相关的标头，如 `Proxy-Authenticate` 和 `WWW-Authenticate`。

- *表示标头*

  请求或响应实体主体（内容）通常可以包含表示标头，包括：

  - 关于实体正文自身的标头，如 `Content-Type`、`Content-Length`、`Content-Location` 和 `Content-Encoding`；
  - 关于实体正文缓存的标头，如 `Expires`。

附录 B 提供了 HTTP 规范中标准标头的完整列表和描述信息。

HTTP 规范仍然在扩展中。像 IETF（Internet Engineering Task Force，因特网工程任务组）或 W3C（World Wide Web Consortium，万维网联盟）这样的组织，可以提议和批准新的标头，扩展 HTTP 协议。HTTP 协议扩展的两个例子是：引入新的缓存标头的 RFC 5861

（http://tools.ietf.org/html/rfc5861）和引入跨源访问标头的 CORS 规范（http://www.w3.org/TR/cors/）。本书后面章节会对这两个扩展进行介绍。

1.2.7　HTTP状态码

HTTP 响应总是返回状态码和描述，说明请求是否成功。这两部分信息由源服务器负责返回，告知客户端服务器是否接受或成功处理了请求，并给出下一步可行操作的建议。描述信息是人工识读的文本，对状态码进行解释。状态码的范围是 4xx~5xx。表 1-2 列出了状态码的不同类别，以及相关的 httpbis 参考文档。

表1-2：HTTP状态码

范围	描　　述	参考文档
1xx	收到请求，正在处理	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-7.2
2xx	收到、接受并理解了请求	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-7.3
3xx	需要更多操作以完成请求	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-7.4
4xx	请求无效，无法完成	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-7.5
5xx	服务器无法完成请求	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-7.6

状态码可能与其他标头直接相关。在下面展示的响应消息片段中，服务器返回状态码 `201`，表示创建了新资源。`Location` 标头向客户端指明了新建资源的 URI，这样，HTTP 客户端在得到状态码 `201` 时应该自动检查 `Location` 信息。

```
HTTP/1.1 201 Created
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Location: http://localhost:8081/api/contacts/6
```

1.2.8　内容协商

HTTP 服务器经常支持同一资源的多种表示方式。资源的表示可能受到多种因素的影响，如客户端的不同功能，或基于有效载荷的优化。例如：对于邮件程序客户端，一个 Contact 资源会返回定制的 vCard 表示。HTTP 允许客户端告知服务器其偏好，参与媒体类型的选择。客户端和服务器之间的这种选择过程称为内容协商（content negotiation，参见 http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-3.4）或 conneg。

1.2.9　缓存

正如我们在“方法属性”部分提到的，有些响应是可缓存的，即 `GET` 和 `HEAD` 请求的响应。缓存的主要好处是可以提高因特网的整体性能和规模。缓存可以通过以下方式为客户端和源服务器提供帮助：

- 由于客户端和服务器之间的往复通信数量减少，响应有效载荷也得以降低，客户端因此受益；
- 中间层可以返回缓存的资源表示，减少了源服务器的负载，服务器因此受益。

HTTP 缓存是一种存储机制，对来自源服务器的缓存响应进行增加、获取和删除等管理。缓存只尝试处理那些使用可缓存方法的请求，所有其他的请求（使用不可缓存的方法）都会自动转发给源服务器。如果请求是可缓存的，但其响应在缓存中不存在或已过期，缓存也会把请求转发给源服务器。

httpbis 定义了一个相当复杂的缓存机制（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-4.1）。HTTP 缓存机制有很多更完善的细节，但基本上是建立于两个概念：过期和验证。

1. 过期

一个响应中，通过 `CacheControl` 标头的 `max-age` 指定了一个时间最大值，如果它在缓存中存在的时间超过这个最大值，那么这个响应就过期了，或者说“陈旧”（stale）了。一个响应的 `Expires` 标头定义了一个过期时间，如果缓存服务器上的当前时间超过了这个过期时间，那么这个响应也会过期。如果一个响应没有过期，那么缓存就可以用它满足请求。不过，在请求和已缓存的响应中，还存在其他一些控制数据（参见下面的“缓存和协商响应”）可能使已缓存的响应不能使用。

2. 验证

如果一个响应过期，那么缓存必须重新对其进行验证。要验证一个响应，缓存会向服务器发送一个条件 `GET` 请求（参见“条件请求”），询问已缓存的响应是否依然有效。所发送的条件请求包含一个缓存验证码，例如：一个 `If-Modified-Since` 标头，包含响应的 `Last-Modified` 值；或者一个 `If-None-Match` 标头，包含响应的 `ETag` 值。如果源服务器认为这个响应仍然有效，就会返回一个不包含主体的响应，状态码为 `304 Not Modified`，以及一个更新后的过期时间。如果这个响应已经改变了，那么源服务器会返回一个新的响应，而这个响应会被最后保存在缓存中，取代当前缓存的资源表示。

使用陈旧的响应

在某些特定的情况下，比如源服务器不可用时，HTTP 会允许缓存使用陈旧的响应。在这种情况下，缓存可以返回陈旧的响应，并在响应中包含一个 `Warning` 标头，以告知用户。Mark Nottingham 提出的“HTTP Cache-Control Extension for Stale Content”草案（参见 http://tools.ietf.org/html/rfc5861），建议使用一个新的 `Cache-Control` 指令（参见“缓存行为”），以解决这些问题。

在验证响应的过程中，客户端可以使用 `stale-while-revalidate` 指令，让缓存返回陈旧

的内容，以避免验证的延迟。如果网络出现故障或者源服务器不可用，客户端可以使用 `stale-if-error` 指令让缓存提供内容。这两个指令命令缓存：如果请求包含这些标头，那么可以提供陈旧的信息。之前提到的 `Warning` 标头则是在告知客户端，缓存中的内容确实是陈旧的。

注意：RFC 5861 被标识为 informational，也就是说尚未标准化，因此并非所有缓存都会对这些附加指令提供支持。

3. 无效

一个响应在缓存后，也可能被设置为无效。一般情况下，如果缓存观察到一个对已缓存资源的请求使用了非安全方法，就会将已缓存的该资源响应设置为无效。如果请求的方法是修改一个资源的状态，那么缓存就知道已缓存的资源表示是无效的。另外，如果观察到的这个非安全请求的响应没有错误，那么缓存也应该将这个请求的 `Location` 和 `Content-Location` 响应设置为无效。

4. 实体标签

实体标签或称 ETag（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-2.3），是当前选定的资源表示在某一个时刻的验证码。实体标签是一个引用的“不透明”的标识符，不应由客户端解析。服务器可以在响应中使用 `ETag` 标头返回实体标签（实体标签也可以缓存）。客户端可以保存实体标签，在未来的条件请求中用做验证码，把实体标签作为 `If-Match` 或 `If-None-Match` 标头的值传递给服务器。请注意：这里提到的客户端也可以是中间层缓存。服务器收到请求后，将请求中的实体标签与服务器上的受请求资源的实体标签进行比较。如果所请求资源在实体标签生成后发生了变化，那么服务器上资源的实体标签也会改变，与请求中的实体标签就不会吻合了。

实体标签分为如下两种（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-2.1）。

- **强实体标签（strong ETag）** 当服务器的资源表示改变时，强实体标签也会随之改变。一个资源强实体标签是唯一的，与同一资源的其他所有表示都不相同（如 `123456789`）。
- **弱实体标签（weak ETag）** 弱实体标签不一定随着资源状态进行更新。一个资源的弱实体标签也不必限定为唯一，不必和同一资源的其他表示不同。弱实体标签必须以 `W/` 开头（如 `W/123456789`）。

实体标签默认是强实体标签，条件请求应优先使用强实体标签。

5. 缓存和协商响应

通过使用 `Vary` 标头（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-4.3），缓存可以提供协商的响应。源服务器可以使用 `Vary` 标头指定一个或多个标头字段，作为执

行内容协商的一部分。如果缓存中的一个资源表示带有 Vary 标头，收到对这个资源的请求时，Vary 标头中所有的字段都必须与请求中的字段相符，才有资格使用这个资源表示。

下面例子中的响应使用了 Vary 标头，指定使用 Accept 标头：

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 183
Vary: Accept
```

6. 缓存行为

请求或响应通过 Cache-Control 标头（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.2l），向缓存机制说明自己的缓存特性。缓存行为可以由源服务器在响应中提供，也可以由客户端在请求中提供。Cache-Control 标头的值是一列缓存指令，说明内容是否可缓存、可以在何处存储、过期策略，以及什么时候应该重新验证或从源服务器重新加载。例如：no-cache 指令要求缓存每次在提供已缓存的响应前都必须重新验证。

Pragma 标头（参见 http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.4）也可以指定 no-cache 值，与 Cache-Control 标头的 no-cache 指令效果相同。

下面给出的响应示例使用了 Cache-Control 标头。在这个示例中，Cache-Control 标头指定了缓存有效期为从 Last-Modified 时间之后 3600 秒（即 1 小时），还指定了在已缓存的表示过期后，缓存服务器必须向源服务器重新验证，然后才能提供内容。

```
HTTP/1.1 200 OK
Cache-Control: must-revalidate, max-age=3600
Content-Type: application/json; charset=utf-8
Last-Modified: Wed, 26 Dec 2012 22:05:15 GMT
Date: Thu, 27 Dec 2012 01:05:15 GMT
Content-Length: 183
```

附录 D 详细介绍了缓存的行为机制。如果你想大致了解 HTTP 缓存，请阅读 Ryan Tomayko 的“Things Caches Do”（http://tomayko.com/writings/things-caches-do）和 Mark Nottingham 的“How Web Caches Work”（https://www.mnot.net/cache_docs/#WORK）。

1.2.10 身份验证

HTTP 为服务器提供一个可扩展框架，帮助其保护资源，并让客户端通过身份验证机制访问服务器。服务器可以保护一个或多个资源，每个资源都分配到一个逻辑分区，即域（realm）。每个域都有各自的*身份验证方案*（authentication scheme），或者支持的授权方式。

服务器在收到访问受保护资源的请求时，会返回一个状态为 401 Unauthorized 或 403 Forbidden 的响应，这个响应还包含一个带有*质询*（challenge）的 WWW-Authenticate 标头，说明客户端必须要经过身份验证才能访问所请求的资源。质询是一个可扩展令牌，描述身

份验证方案和附加的认证参数。例如：如果要访问一个受保护的联系人资源，用于说明使用 HTTP 基本认证方案的质询就应该是：`Basic realm="contact"`。

附录 E 详细介绍了质询 – 响应机制的工作机制。

1.2.11　身份验证方案

前一节提到了身份验证框架，RFC 2617（参见 http://www.ietf.org/rfc/rfc2617.txt）定义了两种具体的身份验证机制：

- 基本认证

  在基本认证方案中，用户名和密码使用 Base64 编码，以冒号间隔，通过明文发送用户身份凭据。因为其自身的不安全性，基本认证（参见 http://tools.ietf.org/html/rfc2617#section-2）通常和 TLS（Transport Layer Security，传输层安全协议）结合在一起使用，即为超文本传输安全协议（HTTPS）。基本认证非常容易实现和访问（包括从浏览器客户端进行访问），这一优点使得很多 API 开发者选择使用基本认证。

- 摘要认证

  使用摘要认证（参见 http://tools.ietf.org/html/rfc2617#section-3）时，用户的身份凭据通过明文发送。摘要认证使用客户端发送的一个校验和（MAC，Message Authentication Code，消息校验码），供服务器验证用户凭据。但是，摘要认证有一些安全和性能缺陷，不常使用。

下面的示例是试图访问受保护资源后得到一个 HTTP 基本认证的质询响应。

```
HTTP/1.1 401 Unauthorized
...
WWW-Authenticate: Basic realm="Web API Book"
...
```

如你所见，服务器返回了一个 `401` 状态码，响应中包含一个 `WWW-Authenticate` 标头，说明客户端必须使用 HTTP 基本认证方案进行认证：

```
GET /resource HTTP/1.1
...
Authorization: Basic QWxpY2U6VGhlIE1hZ2ljIFdvcmRzIGFyZSBTcXVlYW1pc2ggT3NzaWZyYWdl
```

之后，客户端返回原来的请求，并加入 `Authorization` 标头，以访问受保护的资源。

1.2.12　附加身份验证方案

在 RFC 2617 制定之后，又出现了一些新的身份验证方案，包括一些厂商相关的机制。

- AWS认证

  这种方案用于 Amazon Web 服务 S3（http://amzn.to/rest-services）身份验证。客户端把

请求的几个部分连缀成一个字符串，然后用户使用自己的 AWS 共享访问密钥计算出一个 HMAC（Hash Message Authentication Code，散列消息校验码），用于对请求字符串生成签名。

- Azure存储

  Windows Azure 提供几种不同的访问 Windows Azure 存储服务（http://msdn.microsoft.com/library/azure/dd179428.aspx）的身份验证方案，每一种都需要使用共享密钥对请求生成签名。

- Hawk

  这一新方案（https://github.com/hueniverse/hawk）由 Eran Hammer 提出，提供了类似 AWS 和 Azure 的一种通用的共享密钥身份验证机制。密钥不会在请求中直接使用，而是用于计算请求中的 MAC 值。这样可以防止 MITM（Man-In-The-Middle，中间人攻击）劫持密钥。

- OAuth 2.0

  使用这种方案（http://tools.ietf.org/html/rfc6749），资源所有者（即用户）可以把从资源服务器访问受保护资源的许可委托给一个客户端。一个认证服务器给这个客户端颁发一个受限访问令牌，客户端可以使用这个令牌访问资源。这种方案有一个显而易见的优点：用户凭据不会直接发送给试图访问资源的客户端应用程序。

关于 HTTP 身份验证机制和实现方法（包括 OAuth），请详见第 15 章和第 16 章。

1.3 小结

本章大致介绍了 HTTP 的相关背景和概念。本章所介绍的概念并不全面，只为帮助你初步涉猎 HTTP 这个资源库，为 ASP.NET Web API 开发打下基础。你会发现，这里讨论的每个术语都提供了进一步的参考信息。当你真正开始进行 Web API 开发时，这些参考资料会有相当的价值，因此，将它们收藏在你的“后备箱”中吧。让我们接着学习 Web API！

第2章 Web API

Web API 不只是返回 JSON 有效载荷。

在前一章，我们了解了 Web 以及 HTTP（驱动 Web 的应用层协议）的核心内容。本章将讲述 Web API 的演化，介绍各种与 Web API 有关的概念，并讨论不同的 API 风格以及设计 Web API 的方法。

2.1 什么是Web API

Web API 是一个编程接口，用于操作可通过标准 HTTP 方法和标头访问的系统。Web API 可供各种 HTTP 客户端使用，如浏览器和移动设备，并可以使用 Web 基础设施提供的服务，如缓存和并发。

2.2 SOAP Web服务

SOAP（Simple Object Access Protocol，简单对象访问协议）服务不支持在 Web 上使用。HTTP 客户端，如浏览器或类似 curl（参见 http://curl.haxx.se/）这样的工具，调用 SOAP 服务很困难。SOAP 请求必须用 SOAP 消息格式正确编写。客户端必须能够访问一个 WSDL（Web Service Description Language，Web 服务描述语言）文件（这个文件描述了 SOAP 服务提供的操作），还要知道如何正确编写 SOAP 消息。这些限制说明，使用 SOAP 服务与系统进行交互的语义是构建在 HTTP 之上的，而不是以其自身为第一级的。其次，SOAP Web 服务通常要求所有的交互都通过 HTTP `POST` 方法进行，因此，响应消息不能加以缓存。再次，SOAP 服务不允许访问 HTTP 标头，这极大地限制了客户端，使其

无法利用 HTTP 的一些功能，如乐观并发（optimistic concurrency）和内容协商（content negotiation）。

2.3 Web API的起源

2000 年 2 月，Salesforce.com 发布了一个新 API（参见 http://history.apievangelist.com/），允许顾客在自己的应用程序里直接使用 Salesforce 的功能。

随后，在同年 11 月，eBay 发布了一个新的 API（参见 http://investor.ebay.com/common/mobile/iphone/releasedetail.cfm?ReleaseID=28230&CompanyID=ebay），允许开发者利用 eBay 的基础设施，开发电子商务应用程序。

这些 API 和 SOAP API（另一个新的趋势）有何不同？这些 Web API 针对的是第三方用户，以支持 HTTP 的方式加以设计。当时，传统的 API 多数是基于 SOAP 协议，为系统集成设计的。这些新 API 以纯旧式的 XML 作为消息交换格式，使用纯旧式的 HTTP 协议，这使它们能为很多客户端（包括简单的 Web 浏览器等）所使用。它们是最早的一批 Web API，之后又出现了很多别的类型。

在 Salesfore 和 eBay 迈出第一步后的几年间，类似的 API 开始出现。2002 年，Amazon 正式推出了 Amazon Web 服务，之后 Flickr 也在 2004 年推出了 Flickr API。

2.4 Web API革命开始

2005 年夏天，ProgrammableWeb.com 上线，其目标是成为所有 API 相关产品的一站式商店。当时其提供的产品目录中，公共 API（SOAP 和非 SOAP 的）有 32 种，相比 2002 年已经有了很大的增长。而在接下来的几年里，公共 API 的数量有了爆炸式的增长。这些 API 的提供者既有业界巨头，如 Facebook、Twitter、Google、LinkedIn、Microsoft 和 Amazon，也有正处于起步阶段的小型创业公司，如 YouTube 和 Foursquare。到了 2008 年 11 月，ProgrammableWeb 产品目录中的 API 达到了 1000 个。四年后，在本文编写时，已经超过了 7000 个，而大约一年前这个数目不过 4000（参见 http://www.programmableweb.com/news/4000-web-apis-whats-hot-and-whats-next/2011/10/03），可见 API 的增长是加速进行的。

显然，Web API 的趋势将会得以延续。

2.5 关注Web

早期的 Web API 未必注意到底层的 Web 体系结构及其设计上的限制，从而导致了一些不良后果，如臭名昭著的谷歌 Web 加速器事件（参见 http://boingboing.net/2005/05/06/google-

accelerator-i.html），导致了顾客数据和资料的丢失。

不过，近年来，随着第三方 API 的使用者和设备数量的快速增长，这种情况得到了改变。各个组织机构意识到，不能再在设计 API 时忽视 Web 体系结构了，因为这样会带来负面影响，使其无法扩大业务规模、支持更多客户，也无法在不影响现有用户的情况下改进 API，所以，必须改变设计 API 的方式。

Web 体系结构和 HTTP 会对 Web API 的设计造成影响，因此，本章余下的部分会对此进行简要介绍。当你开始使用 APS.NET Web API 开发自己的 Web API 时，这些基础知识也可以帮助你充分利用 Web 的功能。

2.6 Web API指南

这一节列出了一些指导原则，用于区别 Web API 和其他类型的 API。通常来说，关键的不同之处在于，Web API 是支持浏览器使用的。除此之外，Web API 还具有如下特点。

- 可供多种客户端使用（至少支持浏览器使用）。
- 支持标准的 HTTP 方法，如表 1-1 中列举的方法。API 不必使用全部的 HTTP 方法，但是，至少应该支持 GET 以获取资源，还应支持 POST 以进行非安全操作。
- 支持浏览器友好的格式。也就是说，Web API 支持浏览器以及任何其他 HTTP 客户端容易处理的格式。在技术上，浏览器客户端可以使用 XML 栈处理 SOAP 消息，但在格式上需要编写大量的专门用于处理 SOAP 的代码。浏览器容易处理的格式有：XHMTL、JSON 以及 Form URL 编码。
- 支持浏览器友好的认证方式。也就是说，浏览器无需使用特殊的插件或扩展，就可以与服务器进行认证。

2.7 特定领域的媒体类型

前一章介绍了媒体类型的概念。除了我们之前了解到的通用的媒体类型，还有特定领域的媒体类型。这些特定领域的媒体类型带有丰富的与特定应用相关的语义，而 Web API 要进行各种系统交互而非简单的文档传输，因此，这些媒体类型在 Web API 开发中的用处特别大。

vCard（参见 http://tools.ietf.org/search/rfc6350）是一种特定领域的媒体类型，提供一种标准方式，对联系人信息进行电子化描述。很多流行的地址簿和电子邮件应用程序，如 Microsoft Outlook、Gmail 以及 Apple Mail，都支持 vCard 媒体类型。

图 2-1 展示了一个联系人资源的 vCard 表示。

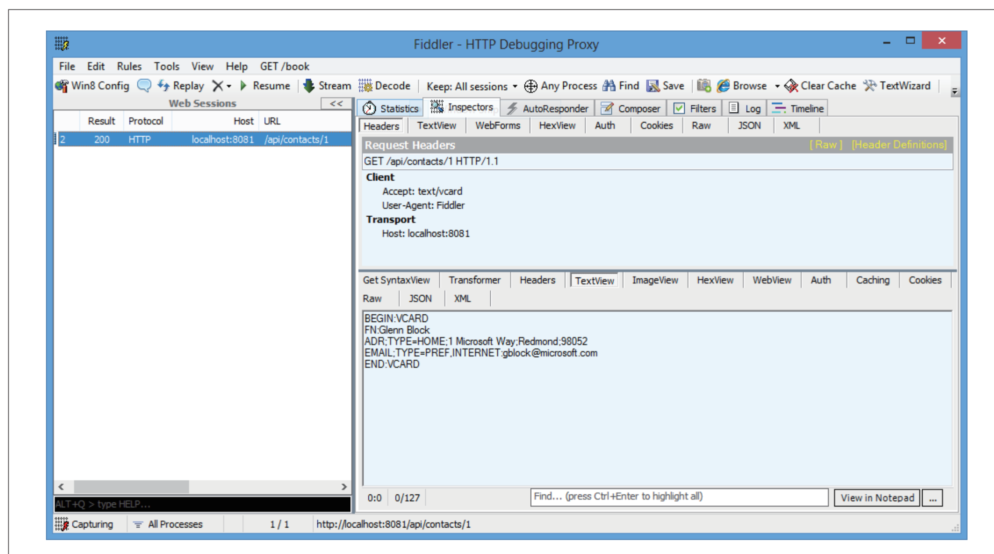


图 2-1：联系人的 vCard 表示

当电子邮件应用程序看到一个 vCard 媒体类型时，它马上就能知道这是联系人信息，应该如何处理。如果这个电子邮件程序是要得到原始 JSON 类型内容，由于 JSON 媒体类型没有定义一个标准方式表明“我是一个联系人”，那么，应用程序就必须等到解析 JSON 内容之后，才知道收到的是什么信息。在这种情况下，格式定义必须在通信之外通过文档来沟通。即使这个格式在文档中定义，也只会在这个应用中使用，而不太可能得到其他的电子邮件应用的支持。而 vCard 是一个标准媒体类型，各种操作系统的诸多应用程序都支持 vCard。

随着应用程序的演化和新需求的出现，我们可以遵循 IANA 注册流程（参见 http://www.iana.org/cgi-bin/mediatypes.pl），创建新的媒体类型。我们可以引入新的媒体类型，客户端可以使用这些新的类型。而如前一章介绍的，客户端通过内容协商来选择媒体类型，现存的客户端也不会被影响，这会给我们的系统带来独特的优势。

2.8 媒体类型档案

如果媒体类型在很多客户端和服务器中使用，就应该在 IANA 中注册。但是如果一个媒体类型用途并不广泛，只在一个应用程序中用到呢？这样的媒体类型还应该在 IANA 中注册吗？有些人认为应该，但是，有些人正在尝试使用更为轻量的、特别是用于 Web API 的机制。媒体类型档案使服务器可以利用现有的媒体类型（如 XML、JSON 等），并提供包含特定应用语义的附加信息。

使用档案链接关系（profile link relation，参见 https://tools.ietf.org/html/rfc6906），服务器可以在 HTTP 响应中返回一个档案链接。链接（link）是一个元素，其中至少包含两个信息：描

述这个链接的 `rel`（或关系）和一个 URI。对于档案链接，链接的 `rel` 值为 `profile`。链接中的 URI 不一定是可参引的（即可访问资源的），但很多时候这个 URI 会指向一个文档。

在今天，档案的使用存在着一个问题：很多媒体类型都不支持表达链接，因此，即便内容中包含了档案信息，也不太可能为客户端所识别。例如：JSON 格式在 Web API 中很常用，却不支持链接。

幸好，任何 HTTP 响应都可以使用其预先制定的*链接标头*（link header，参见 http://tools.ietf.org/html/rfc5988）来传递档案。

在前面介绍的联系人例子中，我们可以返回链接标头，告诉客户端响应中的内容不是任意一个原始 JSON，而是用于 example.com 的联系人管理系统的 JSON 格式。如果客户端在浏览器中访问链接中的 URI，就可以得到描述有效载荷的文档。这个文档可以是任何格式的，例如，新近出现的 ALPS（Application-Level Profile Semantics，*应用级档案语义*，参见 http://alps.io/spec/）格式，就是专为编写档案文档而设计的。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Link: <http://example.com/contactmanagement/profile>; rel="profile"
Date: Fri, 21 Dec 2012 06:47:25 GMT
Content-Length: 183

{
  "contactId":1,
  "name":"Glenn Block",
  "address":"1 Microsoft Way",
  "city":"Redmond","State":"WA",
  "zip":"98052",
  "email":"gblock@microsoft.com",
  "twitter":"gblock",
  "self":"/contacts/1"
}
```

2.9 多个表示

一个资源可以有多个表示，每个表示有着不同的媒体类型。为了说明这一点，让我们来看同一个联系人资源的两种不同表示。图 2-2 展示了第一种 JSON 表示，其中包含了联系人信息。图 2-3 展示的第二种表示是联系人的头像。这两种都是资源状态的有效表示，但有着不同的用途。

客户端会解析 JSON 表示，获取其中数据（如联系人姓名、电子邮件地址等），展示给用户。而 PNG 表示不需要解析，可以直接显示。这是因为 PNG 表示是一个图像，可以很方便地作为 URL 传递，用在 HTML`<img>` 标签里，或者直接用图像查看器显示。正如之前的例子展示的，支持多个资源表示的好处是：具有不同功能的各种客户端都可以与你的 API 进行交互。

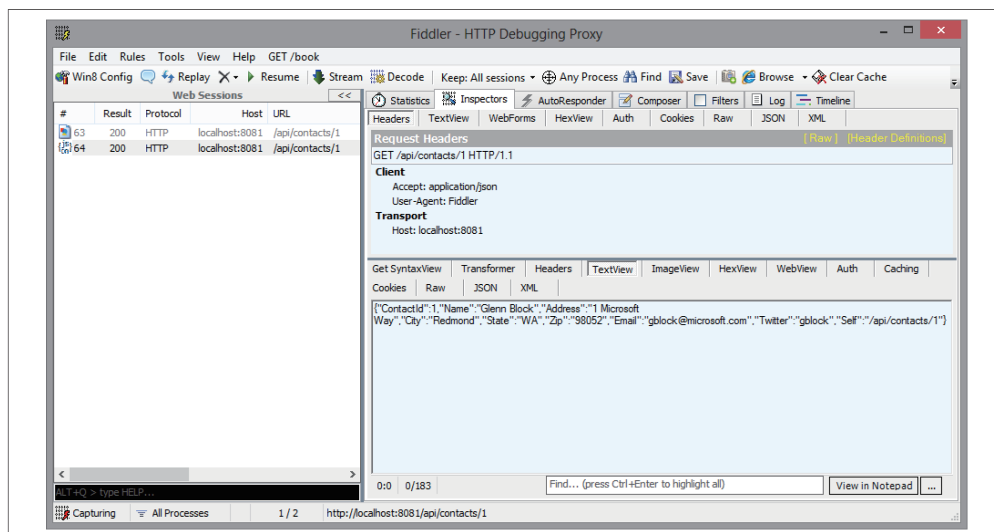


图 2-2：联系人的 JSON 表示

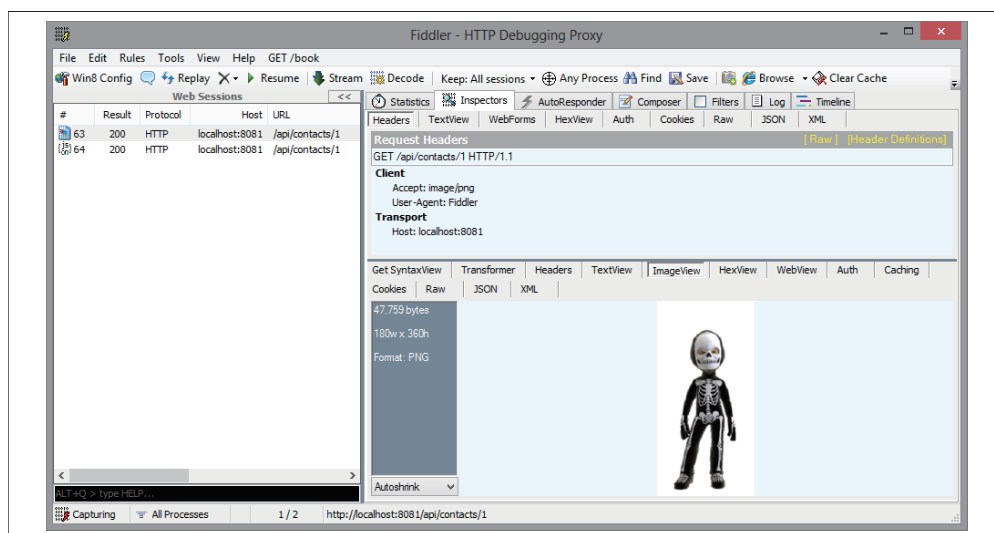


图 2-3：联系人的 PNG 表示

2.10　API风格

Web API 的构建有很多不同的体系结构风格。风格（style）在这里是指在 HTTP 协议上实现 API 的方式。风格是设计中的一组共同特征和约束。每种风格都有各自的利弊。有很重要的一点要注意：风格是 HTTP 的应用，而非 HTTP 自身。

例如，哥特式是一种建筑风格。因为哥特式建筑具有一些共同特征，如尖形拱门、肋状拱顶与飞拱，所以你可以参观不同的建筑，判断哪个具有哥特式风格。同样的道理，同一风格的不同 API 也具有相同的一组特征。现在我们能看到不少 API 风格，但是大致可分为两类：RPC 和 REST。

2.10.1　Richardson成熟度模型

Leonard Richardson 提出的 RMM（Richardson Maturity Model，Richardson 成熟度模型，参见 http://www.crummy.com/writing/speaking/2008-QCon/act3.html）引入了对 API 进行分类的一个框架，根据 API 如何利用 Web 技术，将它们分成不同的级别。

第 0 级，面向 RPC　一个 URI，支持一个 HTTP 方法。
第 1 级，面向资源　很多 URI，支持一个 HTTP 方法。
第 2 级，HTTP 动词　很多 URI，每个 URI 支持多个 HTTP 方法。
第 3 级，超媒体　资源描述自身功能和交互方法。

RMM 模型最初被设计来对当时存在的 API 进行分类，之后变得极为流行。今天，API 社区的很多开发者都使用 RMM 模型来对自己的 API 进行分类。

但是，这个模型也不是完美的。这个模型没有建立一个评分标准来衡量一个 API 符合 REST 的程度。不幸的是，很多人专门利用这一点，把 RMM 模型当作武器，攻击别人的 API 不够符合 REST 标准。这似乎是连 Leonard Richardson 自己也不再推广 RMM 模型的原因之一。

本章，你将进一步了解 RMM 模型的不同级别及其真实的例子。我们将使用 RMM 级别来讨论 API 设计方式的利弊。

2.10.2　RPC（RMM第0级）

第 0 级的 API 使用的是 RPC（Remote Procedure Call，远程过程调用）风格。这种风格的 API 基本上把 HTTP 当作一个传输层协议，用来调用远程服务器上的功能。在 RPC 风格的 API 中，API 在消息的有效载荷中加入自己的语义，用不同的消息类型大致对应远程对象的不同方法，并只使用一个 HTTP 方法：`POST`。第 0 级的 API 的例子有：SOAP 服务、XML-RPC 和 POX（plain old XML）。

让我们来看一个使用 POX 的订单处理系统示例。这个系统提供了一个单一订单处理服务，服务的 URL 为：/orderService。每个客户端向这个服务发出包含不同类型的消息，以便与其进行交互。

为创建一个订单，客户端会发送如下请求：

```
POST /orderService HTTP 1.1
Content-Type: application/xml
Content-Length: xx

<createOrderRequest orderNumber = "1000">
</createOrderRequest>
```

随后服务器发回响应，告知客户端订单创建成功：

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx

<createOrderResponse>
Order created
</createOrderResponse>
```

请注意：创建订单操作的状态不是通过 HTTP 状态码返回的，而是写在正文中的。这是因为 HTTP 只是用做方法调用的传输载体，所有的数据都在有效载荷中发送。

客户端会发送一个 `getOrders` 请求，以获取有效订单列表：

```
POST /orderService HTTP 1.1
Content-Type: application/xml
Content-Length: xx

<getOrdersRequest status="active"/>
</getOrderRequest>
```

服务器的响应包含订单列表：

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx

<getOrdersResponse>
  <orders>
    <order orderNumber = "1000" status="active"/>
    <order orderNumber = "1200" status="active"/>
  </order>
</getOrdersResponse>
```

要批准一个订单，客户端会发送一个 `approveOrder` 请求：

```
POST /orderService HTTP 1.1
Content-Type: application/xml
Content-Length: xx

<approveOrderRequest orderNumber = "1000">
</approveOrderRequest>
```

服务器发回响应，给出批准的结果：

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx

<approveOrderResponse>
  <error code="100">Order approval failed</error>
  <reason>Missing information</reason>
</approveOrderResponse>
```

和之前提到的创建订单的状态相似，此处的错误码也写在消息的有效载荷中。

从这个示例可以看出，RPC 风格的 API 使用有效载荷描述一组待执行的操作及其结果。客户端清楚地知道与每个“服务”相关的各种消息类型，并使用这些消息与系统进行交互。

你可能会问：为什不使用其他 HTTP 方法，如 `PUT` 呢？原因是：使用这种方式，无论进行何种操作，所有的请求都会发送到同一个端点（/orderService）。`POST` 既不是安全方法，也不具有幂等性，因而是 HTTP 中定义限制最少的一种方法。而其他的每种方法都有额外的限制条件，无法满足所有操作的要求。

这种 API 设计方式有一个好处：非常简单，容易实施，也非常符合已有的开发思维模式。

2.10.3　资源（RMM第1级）

在 RMM 第 1 级，API 划分为几个资源，每个资源通过一个 HTTP 方法访问，而每个资源的 HTTP 方法可能不同。与第 0 级不同，第 1 级 API 中的资源 URI 表示的是操作。

让我们回到前面的订单处理示例，看看使用第 0 级的 API 需要发送何种请求。客户端为了创建一个订单，需要向 `createOrder` API 发送一个请求，在有效载荷中传入订单信息：

```
POST /createOrder HTTP 1.1
Content-Type: application/json
Content-Length: xx

{
  "orderNumber" : "1000"
}
```

之后，服务器发回响应，其中包含一个有效的订单信息：

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: xx

{
  "orderNumber" : "1000",
  "status" : "active"
}
```

客户端向 `listOrders` 发送一个请求，指定查询条件，以便获取订单信息。请注意：为进行获取操作，客户端实际发送的是 `GET` 请求，而不是 `POST` 请求。

```
GET /listOrders?status=active
```

服务器返回订单列表：

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: xx

{
  [
    {
      "orderNumber : "1000",
      "status" : "active"
    },
    {
      "orderNumber" : "1200",
      "status" : "active"
    }
  ]
}
```

客户端向 `approveOrder` 资源发送一个 `POST` 请求，来进行订单批准操作：

```
POST /approveOrder?orderNumber=1000
...
```

使用这种风格的一个常见 API 是 Yahoo 的 Filckr API（http://www.flickr.com/services/api/）。阅读其文档，我们可以看到“API Methods”。在 galleries 分类下，列举的方法如图 2-4 所示。

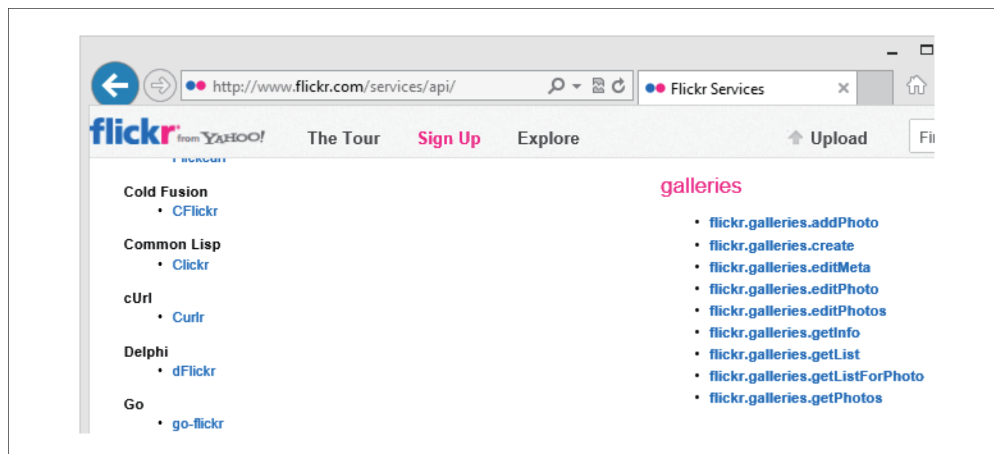


图 2-4：Yahoo Flickr API

有几个不同的 URI 可用来处理照片。要添加照片，你需要向 addPhoto API 发出请求，而获取照片时可以使用 getPhotos。要更新照片，你可以使用 editPhoto 或 editPhotos API。

请注意：在这种风格的 API 中，每个资源会和服务器端对象的一个方法进行对应，而这还是非常类似 RPC 风格的。不同的是，因为第 1 级 API 可以使用 `PUT` 之外的 HTTP 方法，所以有些资源可以通过 `GET` 访问，其响应也可以如之前的示例 `listOrders` 那样进行缓存。使用这种风格还可以得到额外的好处，即可以方便地通过增加资源给系统添加新的功能，而无需更改已有资源，以免导致现有客户端无法工作。

2.10.4　HTTP谓词（RMM第2级）

在之前的示例中，每个资源都与服务器端对象的一个或多个方法相关，和服务器上的具体实现紧密联系在一起，因此，这些示例都是面向功能的（如 `getOrder`）。第 2 级系统使用的是面向资源的方式。API 提供一个或多个资源（如 `order`），每个资源各自支持一个或多个 HTTP 方法。这种风格的 API 在 HTTP 上提供更为丰富的交互方式，支持如缓存和内容协商等功能。

第 2 级 API 通常会区分集合资源（collection resource）和个体资源（item resource）。

- *集合资源*对应子资源的集合（如 http://example.com/orders）。要获取集合，客户端可以向集合资源发送一个 `GET` 请求。要向集合添加一个新成员，客户端可以向集合资源发送一个 `POST` 请求。
- *个体资源*对应集合中的单个子资源（如 http://example.com/orders/1 对应订单 1）。要更新个体资源，客户端可以发送一个 `PUT` 或 `PATCH` 请求。要删除个体资源，客户端可以使用 `DELETE` 方法。使用 `PUT` 方法时，如果待更新的资源不存在，系统通常会允许创建这个资源。个体资源也被泛指为*子资源*（subresource），这是因为 URI 具有层次结构（例如：在 /orders/1 中，1 是 orders 的下一级）。
- 集合资源和个体资源都可以包含一个或多个集合资源和个体资源作为其下一级。

使用这种层次风格来设计订单示例的 API 时，客户端可以发送如下请求进行订单的创建：

```
POST /orders
Content-Type: application/json
Content-Length: xx

{
  "orderNumber" : "1000"
}
```

服务器返回状态码 `201 Created`，响应中的 `location` 标头包含了新创建资源的 URI，还包含一个 `ETag` 标头启用缓存机制：

```
HTTP/1.1 201 CREATED
Location: /orders/1000
Content-Type: application/json
Content-Length: xx
ETag: "12345"

{
  "orderNumber" : "1000",
  "status" : "active"
}
```

为了得到有效订单列表，客户端向 /orders 下的 /active 子资源发送一个 GET 请求：

```
GET /orders/active
Content-Type: application/json
Content-Length: xx

{
  [
    {
      "orderNumber : "1000",
      "status" : "active"
    },
    {
      "orderNumber" : "1200",
      "status" : "active"
    }
  ]
}
```

客户端向 /order/1000/approval 发送一个 PUT 请求，进行订单批准操作：

```
PUT /orders/1000/approval
```

服务器随即发回响应，表明此次的订单批准请求没有成功：

```
HTTP/1.1 403 Forbidden
Content-Type: application/json
Content-Length: xx

{
  "error": {
    "code" : "100",
      "message" : "Missing information"
  }
}
```

通过上面的操作示例，你可以看到客户端与第 2 级 API 的交互方式有哪些不同。客户端使用不同的 HTTP 方法，向一个或多个资源发出请求来表明操作的意图。

GitHub API 是面向资源 API 的一个真实示例。GitHub API（参见 https://developer.github.com/v3/）为 GitHub 中的每个主要区域定义一个根级的集合资源，包括：Orgs、Repositories、

Pull Requests、Issues，等等。每个集合资源都各有下一级的个体和集合资源。你可以使用标准的 HTTP 方法与每个资源进行交互。

例如，要列出当前认证用户的代码库，我们可以向 repos 资源发送如下请求：

```
GET http://api.github.com/users/repos/ HTTP/1.1
```

要为当前认证用户创建一个新的代码库，我们可向同一个 URI 发出 POST 请求，其中包含一个 JSON 有效载荷，说明要创建代码库的 repo 信息：

```
POST http://api.github.com/users/repos/ HTTP/1.1
Content-Type: application/json
Content-Length:xx

{
  "name": "New-Repo",
  "description": "This is a new repo",
  "homepage": "https://github.com",
  "private": false,
  "has_issues": true,
  "has_wiki": true,
  "has_downloads": true
}
```

2.10.5 以资源为中心的API

设计面向资源的 API 颇具挑战性，因为以名词为中心 / 非面向对象的风格是巨大的范式转换，与过去开发者们使用第四代编程语言来设计过程化或面向对象 API 的风格大相径庭。设计面向资源的 API 这一过程，需要分析客户需要与之交互的系统的核心要素，并以资源的形式加以表示。

API 设计者面对的一个挑战是，当现有的 HTTP 方法不能满足设计需求时，该如何处理？例如，如果有一个 Order 资源，该如何操作批准？需要创建一个 Approval HTTP 方法吗？如果你想做一个良好的 HTTP 公民，答案是最好别这样做，因为客户端和服务器永远也不会料到要处理一个 APPROVAL 方法。要解决这个问题，有以下几个方法可供选择。

- 让客户对资源发送 PUT 或 PATCH 请求，在请求的有效载荷中包含 Approval=True。可以使用 JSON 格式，甚至直接在查询字符串中传入一个 URL 编码的表单值也是可以的，例如：

```
PATCH http://example.com/orders/1?approved=true HTTP/1.1
```

- 把 APPROVAL 定义成一个单独的资源，让客户端对其发送 POST 或 PUT 请求：

```
POST http://example.com/orders/1/approval HTTP/1.1
```

2.10.6　超媒体（RMM第3级）

RMM 模型的最后一级是超媒体（hypermedia）。超媒体是响应中的控件或自解释信息（参见 http://www.amundsen.com/blog/archives/1109），可供用户与相关的资源进行交互，改变应用程序的状态。RMM 将超媒体定义为一个严格的级别，但这个定义有些误导性。超媒体可以存在于 API 中，甚至是面向 RPC 的 API 中。

> **Web 超媒体的起源**
>
> 超媒体和超文本是构成 Web 和 HTTP 基础的两个概念。Tim Berners-Lee 在最初的万维网提案（参见 http://www.w3.org/Proposal.html）中这样定义超文本：
>
> > “超文本以网状节点的方式链接和访问各种类型的信息，以供用户随意浏览。超文本还有可能为很多大型的存储信息（如报表、笔记、数据库、计算机文档以及在线帮助系统等）提供单一的用户界面。”
>
> 基于超文本的这一概念，Berners-Lee 进而提议创建一个新的服务器系统（这一系统如今已演化为万维网）：
>
> > “我们建议实施一个简单的方案，将 CERN 已有的几个通过机器存储信息的不同服务器合并，同时通过实验对信息访问需求进行分析。”
>
> 超媒体的概念是从超文本衍生而来的，从简单的文档扩展到包括图像、音频和视频等内容。Roy Fielding 在他有关网络架构的博士论文（参见 http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm）的第 5 章，讨论 REST（Representational State Transfer，表述性状态转移）时，也用到了超媒体一词。他一开始就将超媒体定义为 REST 的一个核心组件：
>
> > “本章介绍并详细描述了为分布式超媒体系统设计的一种架构风格，即表述性状态转移（REST）。”

超媒体的自解释性信息主要分为两大类：链接（link）和表单（form）。要了解二者各自的作用，先来看一下 HTML。

HTML 具备很多不同的超媒体自解释性信息，包括 `<A href>`、`<FORM>` 和 `<IMG>` 标签。当用户在浏览器中查看一个 HTML 页面时，会通过浏览器看到来自服务器的一组不同的链接。用户通过链接的描述或者一个图像来识别这些链接，然后点击感兴趣的链接。HTML 页面也可以包含表单。如果页面中存在一个表单，如创建订单用的那种表单，用户可以填写表单字段，然后点击“提交”。在这两种情况下，用户都不需要对底层 URI 有任何了解，就可以浏览系统。

同样地，超媒体 API 也可以为非浏览器客户端所使用。与前面所举的 HTML 示例类似，API 的响应中可以有链接和表单，但可用不同的格式进行显示，如 XML 和 JSON。要了解更多关于表单的知识，可以参考 CodeBetter 上的“Hypermedia and Forms”（参见 http://codebetter.com/glennblock/2011/05/09/hypermedia-and-forms/）。

超媒体 API 中的链接至少包含两个组件：

- 相关资源的 URI；
- 标识链接资源和当前资源的关系的 `rel`。

链接中的 `rel`（也可能有其他元数据）是用户代理所关注的标识符。`rel` 表明链接所指的资源与当前资源之间有什么关系。

用 H 因子衡量超媒体的自解释性

Mike Amundsen（http://amundsen.com/）设计了一个计量单位：H 因子，用于衡量媒体类型的超媒体支持度。H 因子分为两组，每组各有其因子。

- 链接支持：
  - [LE] 嵌入链接
  - [LO] 外部链接
  - [LT] 模板查询
  - [LN] 非幂等更新
  - [LI] 幂等更新
- 控制数据支持：
  - [CR] 读取请求的控制数据
  - [CU] 更新请求的控制数据
  - [CM] 接口方法的控制数据
  - [CL] 链接的控制数据

H 因子是一种比较和衡量各种现有超媒体 API 类型的有效方法。

回到订单系统的例子，我们现在可以看看超媒体是如何使用的。在前面提到的每个场景中，客户端都完全了解有哪些 URI。但是，在使用超媒体的情况下，客户端只知道一个“根”URI，它会访问根 URI，以便发现系统中可获取资源的信息。这个 URI 的作用几乎类似于一个主页——实际上，根 URI 就是一个主资源（home resource）。

```
GET /home
Accept: application/json; profile="http://example.com/profile/orders"
```

在上面的请求中，客户端发送了一个包含 orders 档案的 Accept 标头。

```
HTTP/1.1 200 OK
Content-Type: application/json; profile="http://example.com/profile/orders"
Content-Length: xx

{
  "links" : [
    {"rel":"orders", "href": "/orders"},
    {"rel":"shipping", "href": "/shipping"}
    {"rel":"returns", "href": "/returns"}
  ]
}
```

这个主资源包含指向系统其他部分的指针——在这个例子里，即订单、发货和退货。要了解如何与这些链接指向的资源进行交互，编写客户端的开发者可以参考档案 URL 当时指向的档案文档。档案文档声明，如果资源的链接的 rel 为 orders，那么客户端可以向资源发送一个订单的 POST 请求，以创建一个新订单。服务器会试图解析得到正确的元素，因此，客户端请求的内容类型只能是简单的 JSON 格式。

客户端发送的请求如下：

```
POST /orders
Content-Type: application/json
Content-Length: xx

{
  "orderNumber" : "1000"
}
```

服务器发回的响应如下：

```
HTTP/1.1 201 CREATED
Location: /orders/1000
Content-Type: application/json; profile="http://example.com/profile/orders"
Content-Length: xx
ETag: "12345"

{
  "orderNumber" : "1000",
  "status" : "active"
  "links" : [
    {"rel":"self", "href": "/orders/1000"},
    {"rel":"approve", "href": "/orders/1000/approval"}
    {"rel":"cancel", "href": "/orders/1000/cancellation"}
    {"rel":"hold", "href": "/orders/1000/hold"}
  ]
}
```

注意，除了订单细节，服务器还会返回几个专门适用于当前订单的链接：

- "self" 标识这个订单自己的 URL，可以用作这个订单资源的书签标记；
- "approval" 标识用于批准这个订单的资源；
- "cancel" 标识用于取消这个订单的资源；
- "hold" 标识用于暂停这个订单处理的资源。

创建订单成功后，接下来客户端就可以使用批准链接，进行批准操作了。档案文档说明，客户端应该对与 approve 的 rel 相关的 URL 发送 PUT 请求，批准订单。

```
PUT /orders/1000/approval
Content-Type: application/json
```

服务器返回的响应与第 2 级 API 处理订单批准请求的响应相同。

```
HTTP/1.1 403 Forbidden
Content-Type: application/json; profile="http://example.com/profile/orders"
Content-Length: xx

{
  "error": {
    "code" : "100",
    "message" : "Missing information"
  }
}
```

Paypal 最近发布了一个新的支付 API（参见 https://developer.paypal.com/webapps/developer/docs/api/），在其响应中使用了超媒体。

下面是使用 Paypal 新 API 进行支付操作得到的响应片段：

```
"links": [
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/sale/1KE480",
    "rel": "self",
    "method": "GET"
  },
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/sale/1KE480/refund",
    "rel": "refund",
    "method": "POST"
  },
  {
    "href": "https://api.sandbox.paypal.com/v1/payments/payment/PAY-34629814W",
    "rel": "parent_payment",
    "method": "GET"
  }
```

如你所见，这个响应中包含一个前面提到的 "self" 链接，还有提交退款操作的链接和访问上一级支付的链接。请注意：除了标准的 href 和 ref 成员外，每个链接还包含一个方法成员，建议客户端应该使用哪种 HTTP 方法。在这个示例中，Paypal 返回的媒体类型是

`application/json`，意味着客户端无法从响应标头中看出这个响应的有效载荷实际上是一个 Paypal 有效载荷。尽管如此，Paypal 的文档里有对 `rel` 加以说明，于是客户端不需要对 URL 硬编码，就能对系统进行访问和操作。

在前面讨论的每个示例中，客户端都只需要知道一个根 URL 就可以访问 API，获得初始响应。在这之后，客户端就通过服务器提供的各种 URL 来进行操作。服务器可以随意修改这些 URL，向客户端提供定制的链接，而不会影响其他客户端。这是使用超媒体带来的好处。然而，凡事有利必有弊。

超媒体系统的实施难度比别的系统要大得多，而且需要设计者改变思维方式。在实施过程中，系统中会引入更多可变的部分。此外，由于用基于超媒体的方式进行 API 开发是一门新兴的、演变中的“科学”，因此，有时需要进行一些探索性的工作。不过，在很多情况下，特别是在构建需要支持很多第三方客户端的开放式系统时，这些代价都是值得的。

2.10.7 REST

REST，即表述性状态转移，可能是如今在 Web API 开发中受误解最多的术语之一。大多数人把 REST 等同于任何容易在 HTTP 上访问的 API，却完全忘记了 REST 的约束条件。

这个词源自之前提到的 Roy Fielding 的有关网络架构的博士论文。在这篇论文中，Fielding 将 REST 描述为分布式多媒体系统的一种架构风格。也就是说，REST 并不是一种技术、一个框架，也不是一种设计模式。REST 是一种*风格*。实践 REST 并没有唯一正确的方法，因此，REST 风格的系统具有很多不同的“风味”。但是，最重要的一点是，所有的 REST 风格的系统都受到一系列的约束。下一节将对此进行更深入的介绍。

对于 REST 的另一个常见误解是，你*必须*构建一个 REST 风格的系统。然而，事实并非如此。REST 约束是设计来创建一个实现一组特定目标的系统的，特别是一个能够长期演化、支持许多不同的客户端、经受许多不同的变化而仍能支持这些客户端的系统。

2.10.8 REST约束

REST 定义了如下 6 条约束（其中一条是可选的）。

- *客户端–服务器*

  REST 系统设计将用户界面与后端分隔。客户端与服务器无关，于是二者可以独立进行演化。

- *无状态*

  在 REST 风格的系统中，所有的应用系统状态都保存在客户端，并在请求中传送给服务器。这使得服务器不必单独记录每个客户端的状态，就可以得到处理请求所需的所有信息。而从服务器端去除了状态信息的管理，也使得应用规模容易扩展。

- *缓存*

  请求中的数据必须能够识别为可缓存的。这条约束使得客户端和服务器的缓存可以代替源服务器返回已缓存的表示。缓存功能极大地降低了延迟，提高了用户体验的性能，并且由于缓存可以降低服务器的负载，还提高了系统的整体规模。

- *统一接口*

  REST 风格的系统对系统交互使用标准化的接口。

- *资源的识别*

  REST 风格的系统中，交互点是*资源*。此处的资源与我们之前讨论的资源概念一致。

- *自描述的消息*

  每个消息都包含客户端和服务器间进行交互所需的所有信息，包括 URI、HTTP 方法、标头、媒体类型等。

- *通过表示对资源执行的操作*

  前面已经介绍过，一个资源可以有一个或多个表示。在 REST 风格的系统中，资源的状态通过这些资源表示来传递。

- *作为应用状态引擎的超媒体*

  之前我们讨论了超媒体，以及超媒体在驱动应用流中所起的作用。这一模型便是 REST 风格系统的核心组件。

- *分层系统*

  REST 风格系统中的组件是分层的，每个组件各自可以访问系统的有限部分。使用分层系统，可以在遗留客户端和服务器之间引入组件层，利用中间层提供附加的服务，如缓存、实施安全策略、压缩等，以适应客户端和服务器的变化。

- *按需代码*

  客户端可以动态下载代码执行，帮助客户端与系统进行交互。一个常见例子是浏览器中的客户端 JavaScript，它就是按需下载执行的。系统可以增加新的应用代码，提高其演化和扩展能力。不过，因为按需代码会降低可见度，因此这条约束被认为是可选的。

由此可见，构建一个 REST 风格的系统要受到诸多约束，也不一定容易实现。关于 REST，有很多书深入讨论了以上提到的问题。虽然使一个系统符合 REST 风格并不容易，但是，如果系统的需求符合 REST 的设计目标，还是值得做出努力的。考虑到这个因素，这本书不会集中讨论 REST，而是关注系统的可演化能力，以及在构建 Web API 时使用什么技术来实现这个目标。这些技术中的每一种都是通向完全 REST 风格系统的途径，各有优势，能给系统带来益处。

换句话说，我们应该关注系统的具体需求，而非是否可给其贴上 REST 风格的标签。

要了解更多关于 REST 术语的说明，Kelly Sommers 写过一篇不错的博客文章，对此进行了详细的阐述（http://kellabyte.com/2011/09/04/clarifying-rest/）。

要了解 REST 风格和超媒体系统的构建，可以参考 O'Reilly 出版的 *REST in Practice*（http://oreil.ly/rest-practice）和 *Building Hypermedia APIs with HTML5 and Node*（http://oreil.ly/build-hyper）。

2.11　小结

本章，我们了解了 API 的起源，探索了业内 API 的增长，还详细讨论了 API 分类。接下来我们要学习微软对于 API 构建的方案——ASP.NET Web API。下一章将介绍 ASP.NET Web API 框架、它的设计目标，以及如何使用它进行 API 的开发。

第3章

ASP.NET Web API 101

按图索骥更简单。

了解了 Web API 对现代网络化应用的重要性原委之后，本章我们将开始接触 ASP.NET Web API。ASP.NET Web API 及其新的 HTTP 编程模型为构建和使用 Web API 提供了新的功能。我们将首先讨论一些核心的 Web API 目标和支持这些目标的功能，然后了解 ASP.NET Web API 的编程模型，理解 ASP.NET Web API 代码如何使用这些支持核心目标的功能。当然，要完成这些任务，最好的方法就是查看 Visual Studio Web API 项目模板提供的代码。最后，我们要修改默认的模板代码，构建我们的第一个“Hello World”Web API。

3.1 核心场景

与很多技术不同，ASP.NET Web API 有完善的、可访问的历史记录（其中一些记录在 CodePlex 上，http://wcf.codeplex.com/）。从一开始，ASP.NET Web API 开发团队就决定要使开发过程尽可能透明，让最终将使用这一产品构建真实系统的专家们能够提出意见，改进产品的设计。这一产品历史可以精简为 ASP.NET 致力实现的核心目标：

- 第一类 HTTP 编程；
- 对称的客户端和服务器编程体验；
- 对不同格式的灵活支持；
- 告别“尖括号编程”；
- 支持单元测试；
- 多种托管选项。

这些虽然是核心目标，却远远不能概括 ASP.NET Web API 框架提供的全部功能。ASP.NET Web API 将 WCF（Windows Communication Foundation）的精华与其可无限扩展的架构相结合，将客户端支持、灵活的托管模型、ASP.NET MVC（Model-View-Controller，模型 – 视图 – 控制器），与约定优于配置、更好的可测试性、高级功能（如模型绑定和验证）结合在一起。我们将会在本章中看到，ASP.NET Web API 框架既简单易上手，也方便按需定制。

3.1.1　第一类HTTP编程

在构建现代 Web API，特别是为比较简单的客户端（如移动设备）设计 API 时，设计的成功与否经常与 API 的表达能力相关。Web API 的表达能力取决于它将 HTTP 作为应用协议使用得有多好。将 HTTP 作为应用协议使用，远比简单地处理 HTTP 请求和生成 HTTP 响应复杂得多。控制应用程序和底层框架行为的是 HTTP 控制流和数据元素，而不是仅仅（偶然地）通过 HTTP 传输的一些附加数据。例如，用于和一个 WCF 服务进行通信的 SOAP 请求：

```
POST http://localhost/GreetingService.svc HTTP/1.1
Content-Type: text/xml; charset=utf-8
SOAPAction: "HelloWorld"
Content-Length: 154

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
        <HelloWorld xmlns="http://localhost/wcf/greeting"/>
  </s:Body>
</s:Envelope>
```

在这个示例中，客户端向服务器发送一个请求，获取一个友好的问候消息。如你所见，这个请求是通过 HTTP 协议发送的。但是，这个请求与 HTTP 的关系也只是到此为止。这个示例中的 API 没有使用 HTTP 方法（有时称为*谓词*）来表明它向服务请求的动作是什么，而是使用同一个 HTTP 方法——`POST`——发送所有的请求，把应用相关的细节封装在 HTTP 请求的正文和定制的 `SOAPAction` 标头中。正如你可能想到的，服务产生的响应也使用了同样的模式：

```
HTTP/1.1 200 OK
Content-Length: 984
Content-Type: text/xml; charset=utf-8
Date: Tue, 26 Apr 2011 01:22:53 GMT
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <HelloWorldResponse xmlns="http://localhost/wcf/greeting">
      ...
    </HelloWorldResponse>
  </s:Body>
</s:Envelope>
```

和请求消息一样，控制应用的协议元素——即客户端与服务器应用相互理解的方式，不在 HTTP 元素中，而是分别放在请求和响应的 XML 正文中。

在这种方式中，HTTP 没有用于表达应用协议，而是简单地作为载体传送另一个应用协议——在这个示例中是 SOAP。当一个服务需要通过许多不同的协议与相似的客户端通信时，这种方式非常适用，但是，如果一个服务需要通过一种协议与各种不同客户端通信时，这种方式就会产生问题。对于 Web API，这些问题尤为典型，因为它不仅客户端多种多样，客户端和服务之间的通信基础架构（如因特网）也各不相同，规模大且持续变化。在这个世界中，客户端和服务的优化目标不应该是获得协议独立性，而应该是基于一个通用的应用协议创建一流的体验。对于通过 Web 进行通信的应用来说，这个协议就是 HTTP。

ASP.NET Web API 的核心是一组 HTTP 的原始对象，其中最重要的两个是 `HttpRequestMessage` 和 `HttpResponseMessage`。这些对象用于为一个实际的 HTTP 消息提供一个强类型的视图。例如，下面是一个 HTTP 请求消息：

```
GET http://localhost:50650/api/greeting HTTP/1.1
Host: localhost:50650
accept: application/json
if-none-match: “1”
```

假设一个 ASP.NET Web API 服务接收到这个请求，我们可以在一个 Web API 控制器类中，使用类似下面的代码，访问和操作这个请求中的各种元素：

```
var request = this.Request;
var requestedUri = request.RequestUri;
var requestedHost = request.Headers.Host;
var acceptHeaders = request.Headers.Accept;
var conditionalValue = request.Headers.IfNoneMatch;
```

这个强类型模型在 HTTP 之上提供了正确层次的抽象，让开发者可以直接使用 HTTP 请求或响应，不必再处理如解析或生成原始数据等低层问题。

3.1.2　对称的客户端和服务器编程体验

使用 HTTP 对象库构建的 ASP.NET Web API 最吸引人的一点是，HTTP 对象库不仅可以在服务器端使用，而且，还可以在使用 .NET 框架构建的客户端使用。这意味着，这里提到的 HTTP 请求，可以被同样的 HTTP 编程模型类创建，最终会用于响应在 Web API 内的请求。本章后面部分将会对此进行详细介绍。

你将会在第 10 章看到，HTTP 编程模型的功能，远不只是对请求和响应的各种数据元素的简单操作。HTTP 编程模型直接提供一些功能（如消息处理程序和内容协商），以便在客户端和服务器都能加以利用，以传递复杂的客户端 - 服务器交互，与此同时尽量多地重用代码。

3.1.3 对不同格式的灵活支持

第 13 章将对内容协商做更为深入的探讨，但概括地说，它是客户端和服务器协同工作，决定在 HTTP 上交换资源表示时使用的正确格式的一个过程。进行内容协商有几种不同的方式和技术，ASP.NET Web API 默认支持服务器驱动的方式，使用 HTTP Accept 标头，让客户端在 XML 和 JSON 之间做出选择。如果请求中没有指定 Accept 标头，ASP.NET Web API 会默认返回 JSON 格式的内容（和 ASP.NET Web API 框架的大部分功能一样，这种默认行为是可以改变的）。

例如，下面是发送到一个 ASP.NET Web API 服务的请求。

```
GET http://localhost:50650/api/greeting HTTP/1.1
```

由于这个请求中没有 Accept 标头告知服务器客户端一个想要的格式，服务器就会返回 JSON 格式的内容。我们可以在请求中加入一个 Accept 标头，指明正确的 XML 的媒体类型标识符[1]，从而改变服务器使用的内容格式。

```
GET http://localhost:50650/api/greeting HTTP/1.1
accept: application/xml
```

3.1.4 告别"尖括号编码"

随着 .NET 框架的成熟，开发者对 XML 配置的抱怨越来越多。为了实现看似基本甚至默认的场景，开发者要进行大量的 XML 配置。更糟糕的是，由于系统配置事项，如控制运行时加载类型的原因，配置一旦修改，可能会导致编译器无法发现的错误，只有在运行时才显现。一个显著的例子是 ASP.NET Web API 的前身 WCF，有严重的 XML 配置问题。尽管 WCF 对自身需要的配置进行了精简，但是，ASP.NET Web API 团队的方向则截然不同，使用完全基于代码的配置模式。第 11 章将详细探讨 ASP.NET Web API 配置。

3.1.5 支持单元测试

随着 TDD（Test-Driven Development，测试驱动开发）和 BDD（Behavior-Driven Development，行为驱动开发）技术的日益流行，人们对许多流行的服务和 Web 框架也产生了更多的不满。这些框架使用静态的上下文对象、密封的类型以及多层继承树，使得人们很难脱离底层的运行时进行对象的创建和初始化，也不容易用"伪"实例替换对象以更好的进行测试隔离，于是，单元测试也就难以进行。

例如，ASP.NET 非常依赖 HttpContext 对象，类似地，WCF 则依赖 OperationContext（或者 WebOperationContext，具体对象因服务类型而异）。使用这种静态的上下文对象有一个

注 1：公共媒体类型目录由 IANA（http://www.iana.org/assignments/media-types/media-types.xhtml）维护。

根本的问题：静态对象是由各自框架的运行时设置并依赖这些运行时使用的。因此，如果要测试使用这种静态上下文对象的服务，就要启动一个服务宿主，运行这个服务。虽然这种做法通常可以为集成测试所接受，但是，要测试诸如 TDD 这样的驱动开发，需要的是能够快速运行较小的单元测试，这种测试方法就不太适用了。

ASP.NET Web API 框架的目标之一是更好地支持这些开发风格，框架的两个特征实现了这一目标。第一个特征是：ASP.NET Web API 的编程模型与 MVC 框架相同，因而可以利用 MVC 框架在几年前实现的可测试性功能，例如：使用抽象机制，避免使用静态上下文对象；使用封装类，以便在单元测试中提供“伪”实例。

第二个特征是：ASP.NET Web API 是以 HTTP 编程模型为核心建立的。在 HTTP 编程模型中，对象的数据结构简单有效，用户可以创建和配置对象，把对象作为参数传递给操作方法，还可以分析返回的方法，于是，可以编写简单、集中的单元测试。在 ASP.NET Web API 框架的发展过程中，其团队始终关注测试，并因此在 Web API 2 中引入了 `HttpRequestContext`。第 17 章将对这种可测试性做更详细的介绍。

3.1.6　多种托管选项

WCF 虽然有不少缺点，但也有很多优点，其中之一就是“自托管”功能。也就是说，WCF 服务可以在任何进程中运行，例如：Windows 服务、控制台应用程序或 IIS（Internet Information Service）。实际上，这种灵活的托管方式几乎弥补了 WCF 在单元测试方面的不足。

在将 WCF Web API 与 ASP.NET 结合形成 ASP.NET Web API 时，开发团队希望能够保留这种自托管功能，于是，和 WCF 服务一样，ASP.NET Web API 服务也可以在你选择的任何进程中运行。第 11 章将详细介绍托管。

3.2　ASP.NET Web API入门

回顾了 ASP.NET Web API 的一些开发目标后，让我们来具体了解创建 Web API 时需要使用的各种元素。要完成这项任务，最简单的方法就是创建一个全新的 ASP.NET Web API 项目，看看项目模板有哪些自动生成。要创建一个新的 ASP.NET Web API 项目，你可以在“New Project”窗口中，展开“Web”节点，选择“ASP.NET Web Application”（参见图 3-1）。

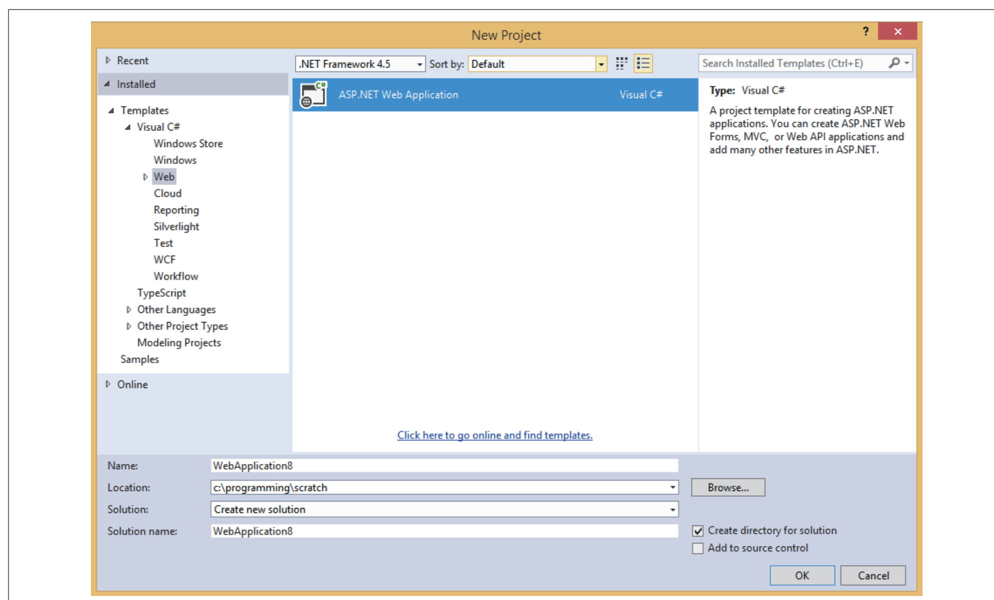


图 3-1：Visual Studio 2012“New Project”窗口中的 MVC4 Web Application 项目

选择了创建 ASP.NET Web 应用程序之后，会出现另外一个对话框，供你选择各种项目配置。在这个对话框中，你可以看到其中一个选项是创建一个 Web API 项目（参见图 3-2）。

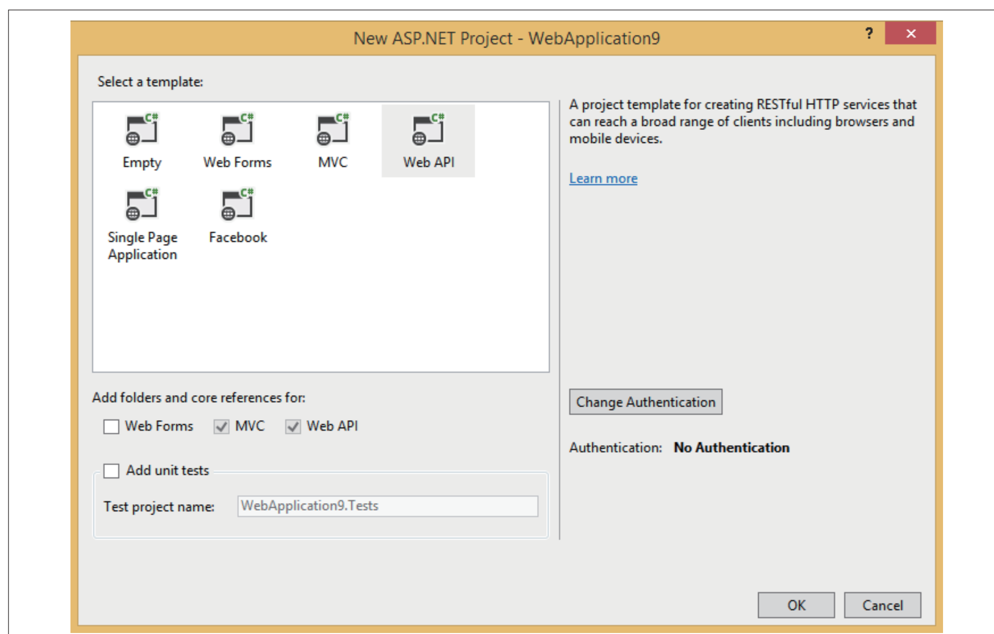


图 3-2：MVC4 New Project 对话框中的 Web API 项目类型

在这个过程中，需要注意的关键一点是，Web API 仅仅是 ASP.NET 项目中的一种项目模板。这意味着 Web API 和其他的 Web 项目类型拥有同样的核心组件，只是在最开始为你创建的模板文件上有所不同而已。这也说明，在图 3-2 中显示的其他任何模板中纳入 Web API 都是合理的，也是人们希望的。

实际上，从本质上说，ASP.NET Web API 只不过是构建在 Web API 框架组件之上、由一个进程托管的一组类，既可以由默认模板设置的 ASP.NET 运行时托管，也可以由你自己定制的宿主托管（本章稍后将详细介绍）。因此，Web API 可以在任何类型的项目中使用，不管是 MVC 项目、控制台应用程序，还是在多个托管项目中引用的类库。

ASP.NET 框架组件是通过 NugGet 软件包管理器（http://docs.nuget.org/）提供给你的 Web API 项目的。表 3-1 中列出了默认项目模板会安装的 NuGet 软件包。要在自己的项目中创建 Web API，你只需确保自己安装了与要使用的功能水平相应的软件包。

表3-1：NuGet的ASP.NET Web API软件包

软件包名称	ID [a]	描　　述	依赖包 [b]
Microsoft .NET Framework 4 HTTP Client Libraries	Microsoft.Net.Http	提供核心 HTTP 编程模型，包括 `HttpRequestMessage` 和 `HttpResponse Message`	无
Microsoft ASP.NET Web API	Microsoft.AspNet.WebApi	NuGet 元包 [c]，提供在 ASP.NET 中安装和托管 Web API 所需安装的所有软件包的一个引用	Microsoft.AspNet.WebApi.WebHost
Microsoft ASP.NET Web API Client Libraries	Microsoft.AspNet.WebApi.Client	包含核心 .NET Framework 4 HTTP 客户端库的扩展，以启用诸如 XML 和 JSON 格式化操作，以及内容协商等功能	Microsoft.Net.Http Newtonsoft.Json [d]
Microsoft ASP.NET Web API Core Libraries	Microsoft.AspNet.WebApi.Core	包含核心的 Web API 编程模型和运行时组件，如关键的 `ApiController` 类	Microsoft.AspNet.WebApi.Client
Microsoft ASP.NET Web API Web Host	Microsoft.AspNet.WebApi.WebHost	包含在 ASP.NET 运行时中托管 Web API 所需的全部运行时组件	Microsoft.Web.Infrastructure Microsoft.AspNet.WebApi.Core

[a] 把软件包 ID 附加到 URL http://www.nuget.org/packages/ 后，可以获得软件包的更多信息。

[b] 当你安装一个软件包时，NuGet 会首先试图安装这个软件包的所有依赖包。

[c] NuGet 元包自身不包含实际信息，只包含其他 NuGet 软件包的依赖信息。

[d] 在由 ASP.NET Web API 使用时，Newtonsoft.Json 是一个外部组件，可以免费下载（http://www.nuget.org/packages/newtonsoft.json）。

除了默认项目模板安装的 NuGet 软件包之外，你也可以安装表 3-2 中列出的 NuGet 软件包。

表3-2：ASP.NET Web API可用的附加NuGet软件包

软件包名称	ID	描　　述	依　赖　包
Microsoft ASP.NET Web API Self Host	Microsoft.AspNet.WebApi.SelfHost	包含在定制进程（如：控制台应用程序）中托管 Web API 所需的全部运行时组件	Microsoft.AspNet.WebApi.Core
Microsoft ASP.NET Web API OWIN	Microsoft.AspNet.WebApi.Owin	提供在 OWIN 服务器中托管 ASP.NET Web API 的功能，以及提供对其他 OWIN 功能的访问	Microsoft.AspNet.WebApi.Core、Microsoft.Owin、Owin

请看 NuGet 软件包的依赖关系图（图 3-3），也许可以帮助你更好地理解如何根据你所设想完成的目标，来决定需要安装哪个或哪些软件包。

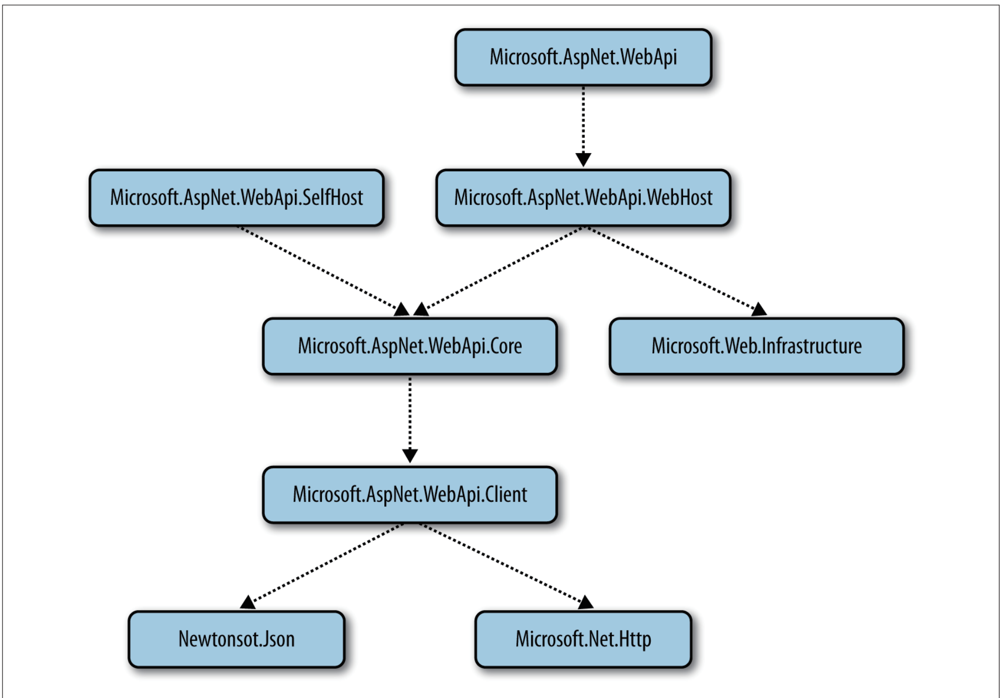


图 3-3：Web API 所需的 NuGet 软件包依赖关系

我们从这个依赖关系图中可以看出，安装任何一个 NuGet 软件包，都会自动安装图中与其直接或间接相连的所有 NuGet 软件包。例如：安装 `Microsoft.AspNet.WebApi` 会自动安装 `Microsoft.AspNet.WebApi.WebHost`、`Microsoft.AspNet.WebApi.Core`、`Microsoft.Web.Infrastructure`、`Microsoft.AspNet.WebApi.Client`、`Newtonsoft.Json` 和 `Microsoft.Net.Http`。

3.3 新建Web API项目

现在我们已经创建了一个新的、Web 托管的 ASP.NET Web API 项目，接下来再看项目模板创建的一些核心元素，我们会将对这些元素进行定制，以便创建自己的 Web API。我们要介绍两个核心文件：WebApiConfig.cs 和 ValuesController.cs（参见图 3-4）。

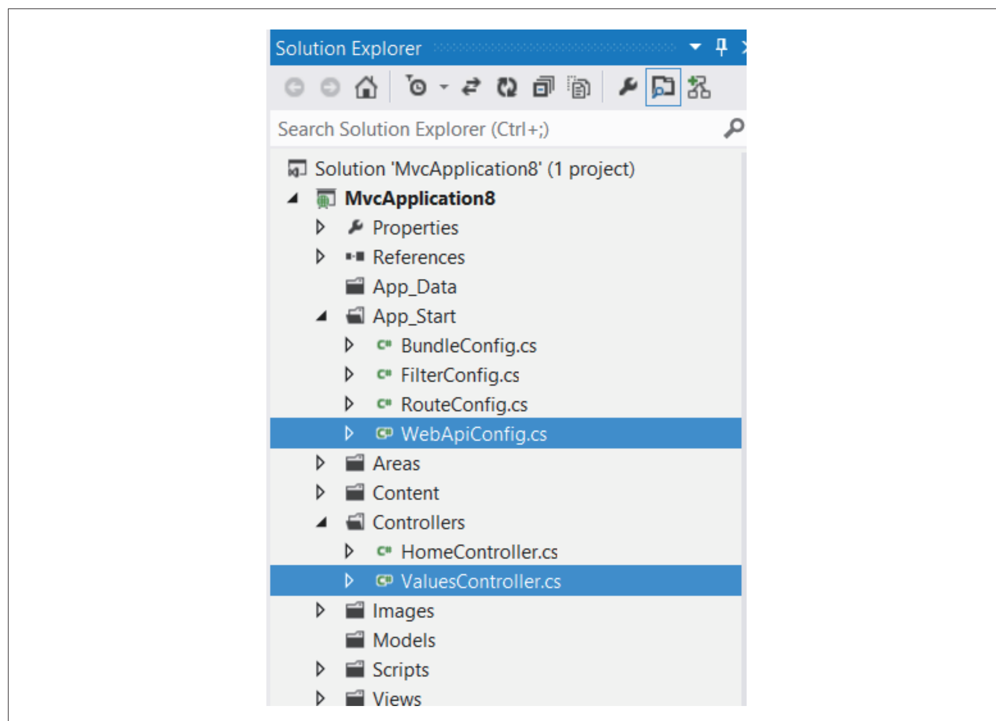


图 3-4：Visual Studio 2013 Solution Explorer 中的 WebApiConfig.cs 和 ValuesController.cs 文件

3.3.1 WebApiConfig

这个 C# 或 Visual Basic.NET 文件位于顶层目录 App_Start 下，并声明了它的 `WebApiConfig` 类。`WebApiConfig` 只包含一个方法 `Register`，由 global.asax 中的 `Application_Start` 方法调用代码。正如 `WebApiConfig` 类的名字表明的，这个类可用于注册 Web API 配置的各个方面。默认情况下，项目模板生成的主要配置代码会注册一个默认的 Web API 路由。这个路由将收到的 HTTP 请求映射到控制器类，并解析 URL 中可能带有的数据元素，确保处理管道中的其他类能够使用这些数据。默认的 `WebApiConfig` 类如示例 3-1 所示。

示例 3-1：默认的 `WebApiConfig` 类

```
public static class WebApiConfig
{
```

```
        public static void Register(HttpConfiguration config)
        {
            // Web API 配置和服务

            // Web API 路由
            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
```

你如果精通 MVC 开发，那么，可能已注意到 ASP.NET Web API 提供了一套用于注册 Web API 路由的扩展方法，与默认的 MVC 路由不同。例如，这个新项目在 `WebApiConfig` 类之外，还包含下面的类：

```
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
                    id = UrlParameter.Optional }
            );
        }
    }
```

一个项目有两个路由注册方法这一点，乍看之下有些让人不知所以，因此，有必要解释一下二者的大致区别。有一点要记住的是，这些“映射”方法只是扩展方法，创建一个路由实例，并把这个实例添加到与宿主相关的路由集合之中。ASP.NET MVC 和 ASP.NET Web API 的区别及其原因，在于它们使用的路由类不同，甚至路由集合的类型也不相同。第 11 章会对这些类型的细节进行更多讨论，但是，ASP.NET Web API 之所以使用与 ASP.NET MVC 不同的路由类型，是为了能够尽量脱离 `System.Web` 程序集里与 `Route` 和 `RouteCollection` 类相关的遗留代码，从而提供更为灵活的托管选项。这种设计带来的直接好处就是，ASP.NET Web API 的自托管能力。

配置 ASP.NET Web API 路由，需要声明 `HttpRoute` 实例并添加到路由集合中。虽然创建 `HttpRoute` 实例的扩展方法和 ASP.NET MVC 中的不同，但是，两个方法的语义几乎一样，都使用相同的元素，如路由名、路由模板和默认参数，甚至都使用路由约束。正如你在示例 3-1 中看到的，项目模板的路由配置代码设置了一个默认的 API 路由，路由的 URI 前缀

为“api”，后面接控制器名和一个可选的 ID 参数。这个路由配置不需要进行任何修改，就足以用来创建提供获取、更新和删除数据功能的 API。这种路由配置之所以如此灵活，是因为 ASP.NET Web API 控制器类把 HTTP 方法映射到控制器的操作方法使其成为可能。本章后文以及第 12 章将更详细地介绍 HTTP 方法映射。

3.3.2 ValuesController

`ApiController`，即 `ValuesController` 的父类，是 ASP.NET Web API 的核心。虽然只需实现 `IHTTPController` 接口的各种成员，就可以创建一个可用的 ASP.NET Web API 控制器，但在实际使用中，大部分 ASP.NET Web API 的控制器还是通过继承 `ApiController` 来创建的。`ApiController` 类负责协调 ASP.NET Web API 对象模型中各个不同的类，在 HTTP 请求的处理中执行一些关键的任务：

- 选择和运行控制器类上的一个操作方法；
- 将 HTTP 请求消息的各元素转换成控制器操作方法的参数，并将操作方法的返回值转换成有效的 HTTP 响应正文；
- 运行各种筛选器，这些筛选器可以是为操作方法或控制器配置的，也可以是全局的；
- 为控制器类的操作方法提供适当的上下文状态。

Web API 模板中的 `ValuesController` 类，继承了 `ApiController` 类并利用其执行的关键处理任务，呈现出 `ApiController` 之上的更高层次的抽象。示例 3-2 展示了 `ValuesController` 代码。

示例 3-2：默认的 `ValuesController` 类

```
public class ValuesController : ApiController
{
    // GET api/values
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    public string Get(int id)
    {
        return "value";
    }

    // POST api/values
    public void Post([FromBody]string value)
    {
    }

    // PUT api/values/5
    public void Put(int id, [FromBody]string value)
    {
```

```
        }

        // DELETE api/values/5
        public void Delete(int id)
        {
        }
    }
```

这个 `ValuesController` 类，虽然简单，却让我们第一次接触到了控制器编程模型。

首先要注意的是控制器中操作方法的命名。默认情况下，ASP.NET Web API 采用传统方式，在一定程度上通过比较 HTTP 方法名和操作方法名来选择执行哪个操作方法。更准确地说，`ApiController` 会寻找名字以相应的 HTTP 方法开头的控制器操作方法。因此，在示例 3-2 中，发送到 `/api/values` 的 HTTP `GET` 请求会触发控制器中无参数的 `Get()` 方法。ASP.NET Web API 框架提供不同的方法定制修改默认的名字匹配逻辑，并提供扩展点，如果你需要的话可以完全替换这套逻辑。第 12 章将详细介绍控制器和方法选择机制。

ASP.NET Web API 除了可以根据 HTTP 方法来选择操作方法，还可以根据请求的其他元素，如如查询字符串参数来进行选择。更重要的是，ASP.NET Web API 框架支持从请求元素到操作方法参数的绑定。默认情况下，ASP.NET Web API 框架结合各种方法的使用来实现参数绑定，采用的算法既支持简单 .NET 类型，也支持复杂 .NET 类型。对于 HTTP 响应，ASP.NET Web API 编程模型允许操作方法返回 .NET 类型，使用内容协商将这些返回值转换成适当的 HTTP 响应消息正文。第 13 章将详细介绍参数绑定和内容协商机制。

到目前为止，我们讨论了一些 ASP.NET Web API 的设计特点，通过项目模板提供的代码，简单了解了其编程模型。接下来让我们更进一步创建第一个 Web API。

3.4 “Hello Web API!”

我们的第一个 ASP.NET Web API 是一个简单的问候服务。在计算机编程文化中，还有什么问候能比“Hello World!”更无处不在呢？因此，我们将从这个简单的只读问候 API 开始，然后通过本章的剩余部分增加一些改进，演示 ASP.NET Web API 编程模型的其他方面。

3.4.1 创建服务

要创建服务，你只需在 Visual Studio 的 New Project 对话框中，选择创建一个新的 ASP.NET Web Application。在 Web Application Refinement 对话框中，选择 Web API。Visual Studio 就会使用默认模板创建一个新的 ASP.NET Web API 项目。

1. 一个只读的问候服务

我们要在默认的 Web API 项目模板中，加入一个新的控制器。要加入新控制器，你可以

添加一个新的类，或者使用 Visual Studio 的控制器模板。如果使用模板添加控制器，你可以在 Solution Explorer 中右键点击 Controllers 目录，在弹出的右键菜单中选择 Add → Controller（参见图 3-5）。

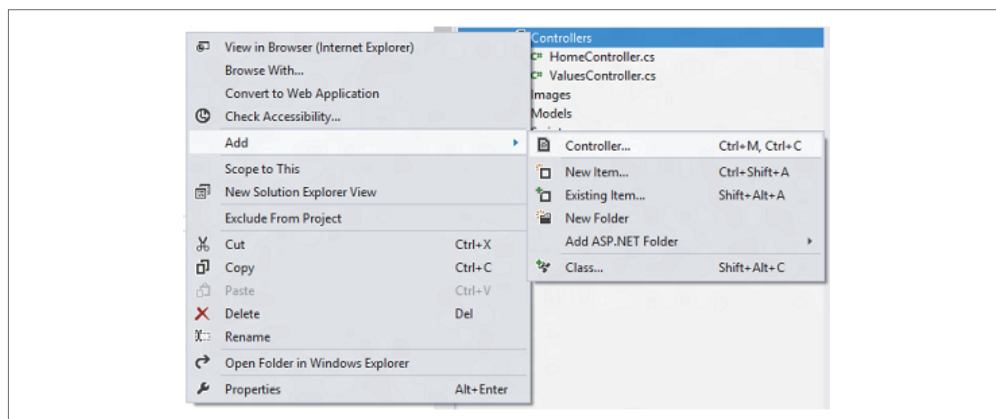


图 3-5：添加新控制器的 Visual Studio 右键菜单

在弹出的对话框中，你可以填入要创建的控制器的更多配置细节。我们要创建的控制器名为 GreetingController，使用 Empty API controller 模板（参见图 3-6）。

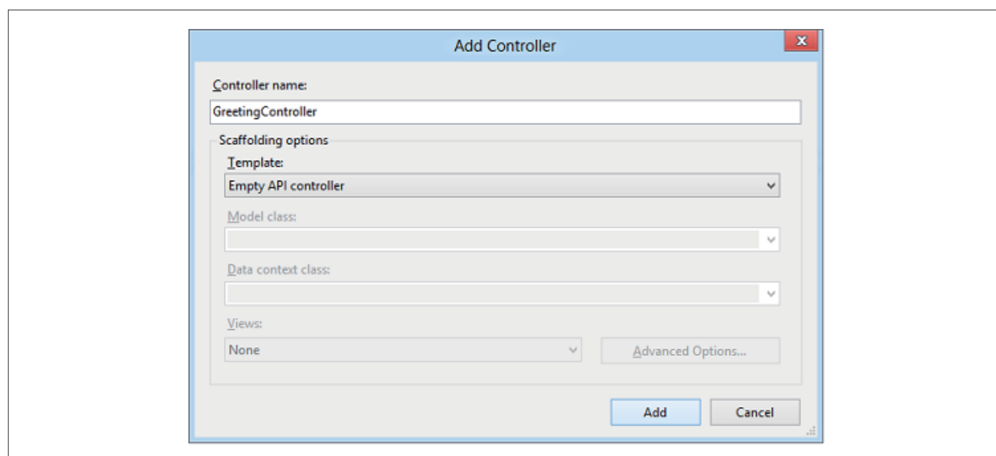


图 3-6：创建 Web API 控制器

填完事项模板对话框之后，Visual Studio 会新建一个 `GreetingController` 类，继承 `ApiController`。为了让我们的新 API 返回一个简单的问候，我们需要给控制器添加一个方法，对 HTTP `GET` 请求做出响应。请记住，按照默认的路由规则，`GreetingController` 会在 HTTP 请求发给 `api/greeting` 时使用。因此，我们要添加一个简单的方法，对 `GET` 请求进行处理：

```
public class GreetingController : ApiController
{
    public string GetGreeting() {
        return "Hello World!";
    }
}
```

现在可以测试一下，看看这个 Web API 是否真的会返回我们的简单问候。要进行测试，我们要使用 HTTP 调试代理工具 Fiddler（http://www.fiddler2.com/）。Fiddler 的一个对 Web API 测试特别有帮助的功能是，它可以构造并执行 HTTP 消息。我们可以使用 Fiddler 的这个功能测试问候 API，操作界面如图 3-7 所示。

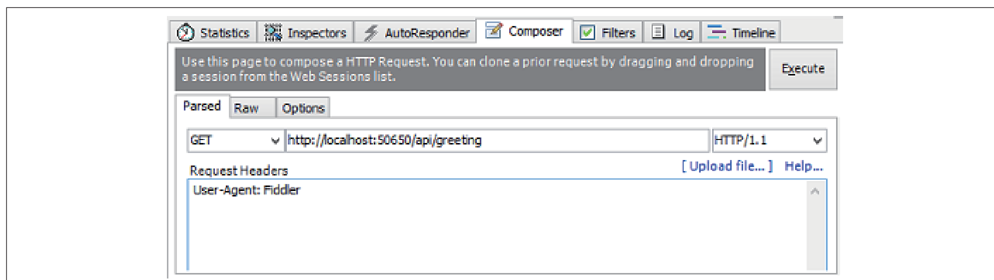


图 3-7：使用 Fiddler 构造一个新的 HTTP 请求

在执行这个请求时，我们可以使用 Fiddler 的会话查看器来浏览请求和响应消息，操作界面如图 3-8 所示。

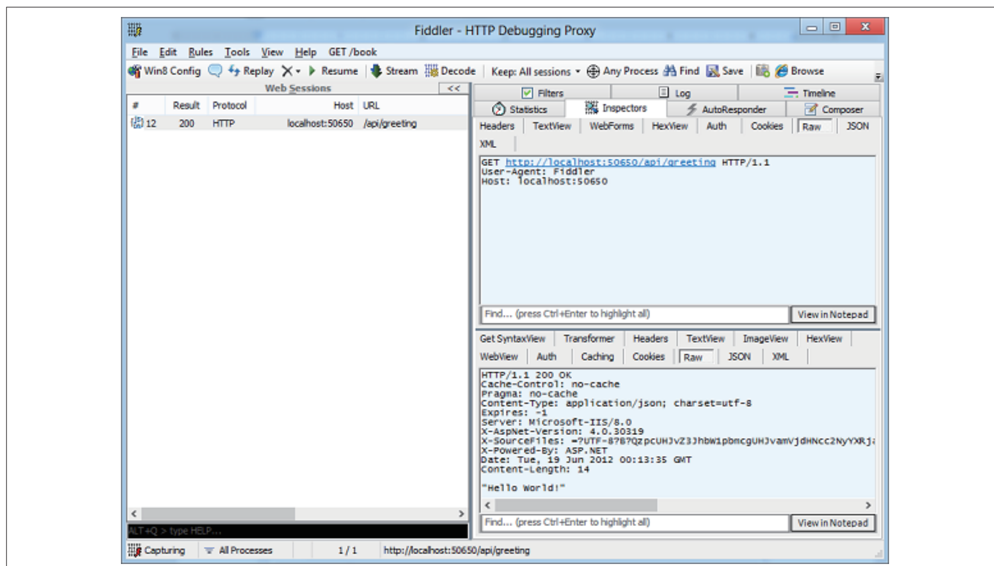


图 3-8：使用 Fiddler 检查 HTTP 请求和响应消息

发送到问候服务的这个简单的 HTTP GET 请求返回了字符串“Hello World!”，测试通过。

2. 内容协商

让我们再回头仔细看一下图 3-8 中，HTTP 响应消息的 Content-Type 标头。在默认情况下，ASP.NET Web API 使用图 3-3 中介绍常见的 Json.NET 类库，将操作方法的返回值转换成 JSON 格式。但是，正如本章之前介绍的，ASP.NET Web API 支持服务器驱动的内容协商，并且，默认支持选择 JSON 和 XML 其中一个。为了了解内容协商的功能，我们回到 Fiddler 的请求消息编辑器，在请求标头文本框中添加一行：

```
accept: application/xml
```

再次执行这个请求，你会看到响应消息现在包含标头 Content-Type: application/xml，现在的响应正文为 XML 格式（参见图 3-9）。

图 3-9：进行内容协商的基本问候 API 的请求和响应消息

3. 添加一个问候

以不同的格式返回一句问候确实很有意思，但是，有实际用途的 API 都需要能够操作系统的状态或数据。因此，我们要扩展问候服务，让用户可以添加新的问候语。用户可以指定一个问候名和问候消息，添加到服务中，之后可以通过一个包含问候名的 URL 来 GET 这个问候消息。另外，如果用户由于拼写错误或其他原因没有在 URL 中指定正确的问候名，我们还需要返回一个 HTTP 404 状态码，告诉用户未找到请求的资源。

要让用户在服务器上创建一个新的问候，我们需要先创建一个模型类，用于保存问候的名字和消息属性。可以在项目的 Models 文件夹中添加如下的类：

```
public class Greeting
{
    public string Name {
        get;
        set;
```

```
    }

    public string Message    {
        get;
        set;
    }
}
```

接下来，我们要在 GreetingController 类中添加一个操作方法，用于处理 HTTP POST 请求，这个方法的参数是 Greeting 实例。

这个操作把参数中传入的问候添加到一个静态列表中，返回一个带着 Location 标头的 HTTP 状态码 201，指向新创建问候的 URL。有了这个 Location 标头，客户端就可以直接通过这个链接值访问新创建的问候资源，而不用自己构建 URL，而因为服务器 URL 结构可能会随时发生变化，使用返回的资源 URL 就会使客户端适应性更强。

```
public static List<Greeting> _greetings = new List<Greeting>();

public HttpResponseMessage PostGreeting(Greeting greeting) {
    _greetings.Add(greeting);

    var greetingLocation = new Uri(this.Request.RequestUri,
        "greeting/" + greeting.Name);
    var response = this.Request.CreateResponse(HttpStatusCode.Created);
    response.Headers.Location = greetingLocation;

    return response;
}
```

向静态集合添加新问候之后，我们创建了一个 URI 实例来表示它的地址，以便在之后的请求中能找到这个新问候。接着，我们使用 HttpRequestMessage 实例的工厂方法 CreateResource（这个方法由基类 ApiController 提供），创建一个新的 HttpResponse Message。在操作方法内部使用 HTTP 对象模型实例，是 ASP.NET Web API 的核心功能之一。使用这一功能，开发者可以对 HTTP 消息元素（如 Loation 标头）进行细粒度的控制，又不依赖于诸如 HttpContext 或 WebOperationContext 这样的静态上下文对象。这一点在提高 Web API 控制器的可测试性方面特别有用，接下来我们会加以讨论。

最后，我们要给 GetGreeting 方法添加一个过载方法，取得并返回客户提供的定制问候：

```
public string GetGreeting(string id) {
    var greeting = _greetings.FirstOrDefault(g => g.Name == id);
    return greeting.Message;
}
```

这个方法很简单，只要找到第一个 Name 属性和传入的 id 参数匹配的问候，然后返回这个问候的 Message 属性。需要注意的是，现在这个方法没有对 id 参数进行任何输入验证。下一节将对此进行讨论。

在默认情况下，HTTP POST 消息的正文由一个 MediaTypeFormatter 对象进行处理，这个对象是根据 Content-Type 请求的标头信息所选择的。相对应地，我们下面展示的这个 HTTP 请求，将由默认的 JSON 格式化程序处理，使用 Json.NET 将 JSON 字符串反序列化，生成 Greeting 类的一个实例：

```
POST http://localhost:50650/api/greeting HTTP/1.1
Host: localhost:50650
Content-Type: application/json
Content-Length: 43

{"Name": "TestGreeting","Message":"Hello!"}
```

随后，这个生成的实例将被传递给 PostGreeting 方法，添加到问候集合中。在 PostGreeting 处理完请求后，客户端将得到如下的 HTTP 响应：

```
HTTP/1.1 201 Created
Location: http://localhost:50650/api/greeting/TestGreeting
```

根据 HTTP 响应中的 Location 标头信息，客户端就可以接着发送请求，访问这个新问候：

```
GET http://localhost:50650/api/greeting/TestGreeting HTTP/1.1
Host: localhost:50650
```

和最开始的只读问候服务一样，客户端可预期返回如下响应：

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Content-Length: 8

"Hello!"
```

4. 错误处理

只要服务器不发生任何错误，并且所有的客户端都遵循同样的规则和约定，前面提到的 HTTP 消息交换就能有效进行。但是，如果服务器出错或者收到无效的请求，会出现什么情况呢？在这方面，创建和使用 HTTP 对象模型实例的能力起到了很大的作用。在示例 3-3 中，我们希望操作方法根据问候名返回其字符串。如果没有找到所请求的问候名，我们希望返回一个带有 HTTP 状态码 404 的响应。为了实现这种功能，ASP.NET Web API 提供了 HttpResponseException 类。

示例 3-3：无法找到问候时返回一个 404 状态码

```
public string GetGreeting(string id) {
    var greeting = _greetings.FirstOrDefault(g => g.Name == id);
    if (greeting == null)
        throw new HttpResponseException(HttpStatusCode.NotFound);
    return greeting.Message;
}
```

在找不到问候名时，直接返回一个新的、包含状态码 404 的 HttpResponseMessage 也是合理的，但是，这种处理方式要求 GetGreeting 操作方法总是返回一个 HttpResponseMessage，会使非异常处理部分的代码路径变得过于复杂，而这是没有必要的。另外，响应消息需要经过整个 Web API 管道，这在异常发生时也很可能是没有必要的。考虑到这些因素，我们选择在操作方法中发出一个 HttpResponseException，而不是返回一个 HttpResponseMessage。如果需要让异常包含一个支持内容协商的响应正文，那么，你可以使用控制器基类的 Request.CreateErrorResponse 方法生成一个 HttpResponseMessage，传递给 HttpResponseException 的构造函数。

5. 测试

直接使用 HTTP 对象模型而非静态上下文对象，带来一个额外的好处，使你可以对 Web API 控制器编写有意义的单元测试。第 17 章将详细介绍有关测试的内容，但是，在这个介绍性的示例中，还是让我们为 GreetingController 的 PostGreeting 操作方法快速编写一个单元测试：

```
[Fact]
public void TestNewGreetingAdd()
{
    // 准备
    var greetingName = "newgreeting";
    var greetingMessage = "Hello Test!";
    var fakeRequest = new HttpRequestMessage(HttpMethod.Post,
        "http://localhost:9000/api/greeting");
    var greeting = new Greeting { Name =
        greetingName, Message = greetingMessage };

    var service = new GreetingController();
    service.Request = fakeRequest;

    // 操作
    var response = service.PostGreeting(greeting);

    // 断言
    Assert.NotNull(response);
    Assert.Equal(HttpStatusCode.Created, response.StatusCode);
    Assert.Equal(new Uri("http://localhost:9000/api/greeting/newgreeting"),
        response.Headers.Location);
}
```

这个测试遵循单元测试编写的标准模式——准备 – 操作 – 断言（arrange，act，assert）。我们创建一些控制状态（包括一个新的 HttpRequestMessage 实例），以表示整个 HTTP 请求；然后使用上下文调用测试的方法；最后对响应进行一些断言。在这个测试中，响应是一个 HttpResponseMessage 实例，由此，我们可以对响应自身的数据元素进行断言。

3.4.2 客户端

正如本章开始提到的，围绕一个核心 HTTP 编程模型构建 ASP.NET Web API，还有一个重要的好处：同样的编程模型也可以用于为客户端和服务器构建极好的 HTTP 应用。例如，我们可以使用如下代码构建一个请求，由第一个 `GetGreeting` 操作方法处理：

```
class Program
{
    static void Main(string[] args)
    {
        var greetingServiceAddress =
            new Uri("http://localhost:50650/api/greeting");

        var client = new HttpClient();
        var result = client.GetAsync(greetingServiceAddress).Result;
        var greeting = result.Content.ReadAsStringAsync().Result;

        Console.WriteLine(greeting);
    }
}
```

和在服务器上一样，这里的客户端代码可以创建和处理 `HttpRequestMessage` 和 `HttpResponseMessage` 实例。另外，ASP.NET Web API 扩展组件，如媒体类型格式化程序和消息处理程序，既可以在服务器端使用，也可以在客户端使用。

3.4.3 宿主

开发一个在传统 ASP.NET 应用中托管的 ASP.NET Web API，和创建任何其他类型的 ASP.NET MVC 应用的方式非常相似。然而，ASP.NET Web API 的一大特点是，它可以托管在你指定的任何进程中，而几乎不需要为此做额外的工作。示例 3-4 展示了将我们的 `GreetingController` 托管于一个定制宿主进程（在这个示例中是控制台应用程序）所需的代码。

示例 3-4：一个简单的 Web API 控制台宿主

```
class Program
{
    static void Main(string[] args)
    {
        var config = new HttpSelfHostConfiguration(
        new Uri("http://localhost:50651"));

        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional });

        var host = new HttpSelfHostServer(config);
```

```
            host.OpenAsync().Wait();

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();

            host.CloseAsync().Wait();
        }
    }
```

如果要用一个定制进程托管 Web API，我们不需要修改控制器，也不需要在 app.config 文件中添加任何神奇的 XML 配置。实际上，我们只要创建一个 `HttpSefHostConfiguration` 实例，配置地址和路由信息，然后开启这个宿主。一旦宿主开启并监听请求，我们就阻塞主控制台线程，从而防止服务器关闭。当用户选择关闭宿主时（按任意键），我们就关闭 Web API 宿主，退出控制台应用程序。第 11 章将详细讨论托管机制。

3.5 小结

本章，我们描述了 ASP.NET Web API 的一些关键设计目标，然后用 Web API 项目模板展示了组成框架的不同组件如何通过 NuGet 组织和发布，并且利用默认的模板代码探索了框架的编程模型。最后，我们编写了“Hello World!”Web API，还使用了 ASP.NET Web API 的自托管功能。

从第 4 章开始，后面的章节将对本章介绍的各个主题逐一进行深入地探讨。第 4 章将探索 ASP.NET Web API 的底层工作机制。

第 4 章

处理架构

现在来讨论些完全不同的东西。

前一章讨论了核心 ASP.NET Web API 的编程模型，介绍了显示在框架中的一系列基础概念、接口和类。在开始讨论这本书的核心主题，即可演化的 Web API 设计之前，本章要绕个弯子先审视其底层机制，探究 ASP.NET Web API 的内部机制，展现底层的处理架构，详细介绍在收到 HTTP 请求和返回相应的 HTTP 响应信息之间发生了什么。本章也可以用作一张路线图，帮你初步了解这本书第三部分将要介绍的 ASP.NET Web API 高级功能。

在本章中，我们将以示例 4-1 的 HTTP 请求，结合示例 4-2 中的控制器作为一个具体例子，说明 ASP.NET Web API 处理架构的运行时行为。ProcessesController 控制器包含一个 GET 操作，这个操作返回一个表示，包含指定图像名称的所有进程。示例中的 HTTP GET 请求的对象是被 http://localhost:50650/api/processes?name=explorer 标识的资源，这个资源代表了所有正在运行的浏览器进程。

示例 4-1：HTTP 请求消息示例

```
GET http://localhost:50650/api/processes?name=explorer HTTP/1.1
User-Agent: Fiddler
Host: localhost:50650
Accept: application/json
```

示例 4-2：控制器示例

```
public class ProcessesController : ApiController
{
    public ProcessCollectionState Get(string name)
    {
```

```
        if (string.IsNullOrEmpty(name))
        {
            throw new HttpResponseException(HttpStatusCode.NotFound);
        }
        return new ProcessCollectionState
        {
            Processes = Process
                .GetProcessesByName(name)
                .Select(p => new ProcessState(p))
        };
    }
}

public class ProcessState
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double TotalProcessorTimeInMillis { get; set; }
    ...

    public ProcessState() { }
    public ProcessState(Process proc)
    {
        Id = proc.Id;
        Name = proc.ProcessName;
        TotalProcessorTimeInMillis = proc.TotalProcessorTime.TotalMilliseconds;
        ...
    }
}

public class ProcessCollectionState
{
    public IEnumerable<ProcessState> Processes { get; set; }
}
```

图 4-1 展示的 ASP.NET Web API 处理架构分为三层。

- *托管*（hosting）层，位于 Web API 和底层的 HTTP 栈之间。
- *消息处理程序管道*（message handler pipeline）层，可以用于实现横切关注点（cross-cutting concern），如日志和缓存。但是，OWIN（http://owin.org/，将在第 11 章中介绍）的引入将消息处理程序管道的一些功能下移到了栈下端的 OWIN 中间件中。
- *控制器处理*（controller handling）层，控制器和操作是在这一层进行调用的，参数在此绑定和验证，HTTP 响应消息也在这创建。此外，这一层还包含和执行*筛选器管道*（filter pipeline）。

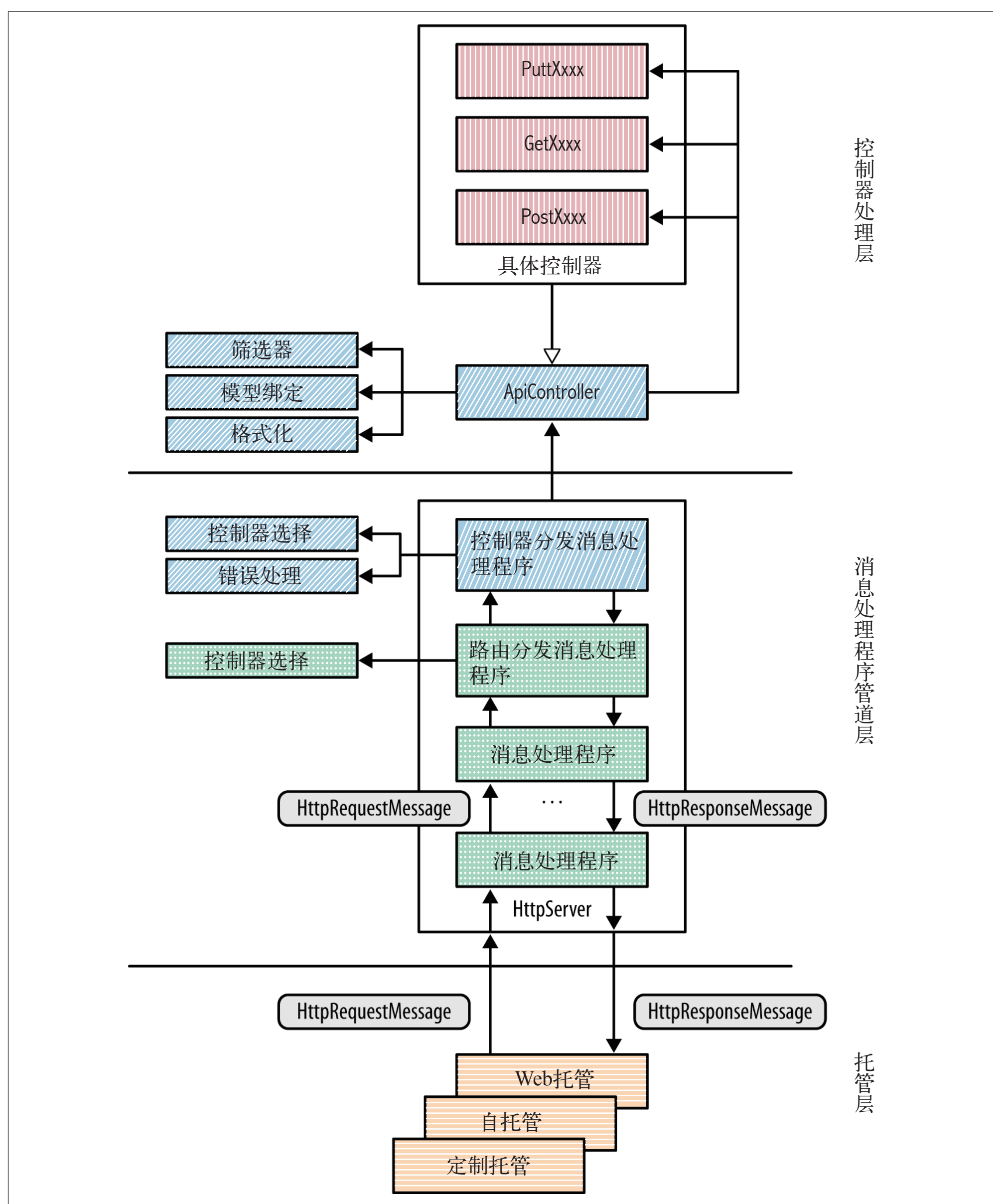


图 4-1：简化的 ASP.NET Web API 处理模型

接下来将对这些层次逐一进行介绍。

4.1　托管层

Web API 处理构架的最底层负责 API 托管，是 Web API 和底层 HTTP 基础结构的接口，例

如：经典的 ASP.NET 管道、.NET Framework 的 System 程序集中的 HttpListener，或一个 OWIN 宿主。托管层负责创建 HttpRequestMessage 实例，表示 HTTP 请求，然后将其推送到上层的消息处理程序管道。在处理响应时，托管层还负责获取从消息处理程序管道返回的 HttpResponseMessage 实例，把它转换成可由底层网络栈处理的响应消息。

请记住，HttpRequestMessage 和 HttpResponseMessage 是 .NET Framework 4.5 中引入的代表 HTTP 消息的新类。这两个新类是 Web API 处理架构的核心，托管层的主要任务就是，在这两个类和底层 HTTP 协议栈使用的本地消息表示之间建立连接。

在我们编写本书时，ASP.NET Web API 支持几种不同的托管层，分别是：

- 在任何 Windows 进程(例如,控制台应用程序或 Windows 服务)中的*自托管*(self-hosting)；
- *Web 托管*（Web hosting），即在互联网信息服务（IIS）之上使用 ASP.NET 管道；
- *OWIN 托管*（使用 OWIN 兼容的服务器，如 Katana）[1]。

第一种托管方法建立在 WCF 的自托管功能之上，第 10 章将对此进行更详细的介绍。

第二种方法——Web 托管——使用了 ASP.NET 的管道和路由功能，将 HTTP 请求转发到一个新的 ASP.NET 处理程序，HttpControllerHandler 中。这个处理程序将收到的 HttpRequest 实例转换成 HttpRequestMessage 实例，然后推送到 Web API 管道，从而在传统的 ASP.NET 管道和新的 ASP.NET Web API 架构间建立起了连接。这个处理程序还负责获取 Web API 返回的 HttpResponseMessage 实例，将其复制为 HttpResponse 实例，然后传递给下层的 ASP.NET 管道。

第三种托管方法，是在一个 OWIN 兼容的服务器上建立一个 Web API 层。使用这种方法时，托管层把 OWIN 上下文对象转换成一个新的 HttpRequestMessage 对象，发送给 Web API。反过来，托管层也会把 Web API 返回的 HttpResponseMessage 写入 OWIN 上下文。

还有第四种方法，就是完全不用托管层：HttpClient 使用与 Web API 同样的类模型，直接发送请求给 Web API 运行时，无需进行任何转换。第 11 章将对托管层进行更深层次的探讨。

4.2 消息处理程序管道

Web API 处理架构的中间层是*消息处理程序管道*。这一层提供了一个扩展点，拦截器可以在这里处理横切关注点，如日志和缓存。这一层在作用上与框架中间件的概念类似，如 Ruby 的 Rack（参见 http://rack.rubyforge.org/doc/SPEC.html）、Python 的 WSGI（Web Server Gateway Interface，Web 服务器网关接口，参见 http://legacy.python.org/dev/peps/pep-3333/）和 Node.js Connect Framework（http://www.senchalabs.org/connect/）。

注 1：ASP.NET Web API 第 2 版中引入了 OWIN 托管支持。

消息处理程序是对一个操作的抽象，它接受 HTTP 请求消息（`HttpRequestMessage` 实例）并返回 HTTP 响应消息（`HttpResponseMessage` 实例）。ASP.NET Web API 消息处理程序管道是这种处理程序的组合，管道中的每个程序（最后一个除外）都有指向下一个程序的指针，这个指针称为*内部处理程序*（inner handler）。使用这种管道组织方式，Web API 能够执行的操作就具有了很大的灵活性，如图 4-2 所示。

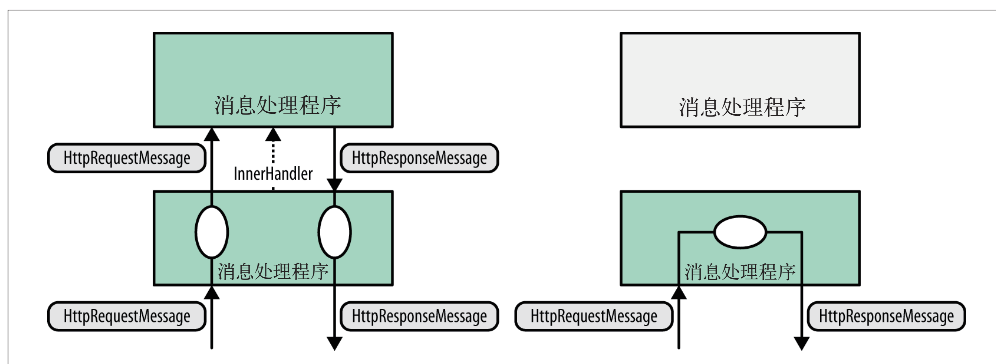


图 4-2：消息处理程序流程示例

图左侧展示了如何使用处理程序，对请求和响应消息分别执行一些预处理和处理后操作。通过 `InnerHandler`，处理流程从一个处理程序移动到另一个，一个方向的流程进行请求消息处理，反方向进行响应消息处理。以下是一些预处理和处理后操作的例子：

- 在控制器的操作方法处理消息之前，根据请求消息是否包含某个标头（如 `X-HTTP-Method-Override`），改变请求的 HTTP 方法；
- 添加一个响应标头，如 `Server`；
- 获取和记录诊断信息或业务指标数据。

你可以使用处理程序直接产生一个 HTTP 响应，提前终止管道流程，如图 4-2 右侧所示。通常采用这种做法的情况是，当请求消息没有得到正确地身份验证，立即返回一个状态码为 `401`（`Unauthorized`）的 HTTP 响应。

在 .NET 框架中，消息处理程序是新抽象类 `HttpMessageHandler` 的派生类，类的层次关系如图 4-3 所示。

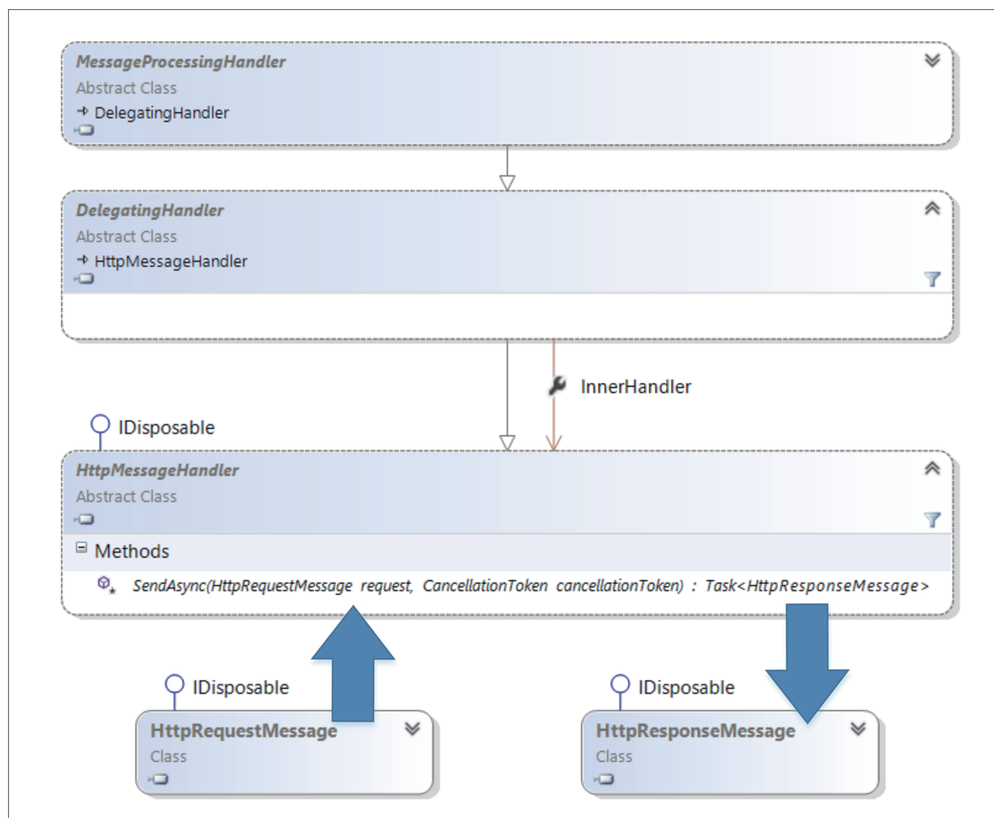


图 4-3：消息处理程序的类层次

抽象方法 SendAsync 接收一个 HttpRequestMessage 实例，通过返回 Task<HttpResponse Message>，异步生成一个 HttpResponseMessage。这个方法遵循 TAP（Task-based Asynchronous Pattern，基于任务的异步模式，参见 http://msdn.microsoft.com/en-us/library/hh873175.aspx）关于取消操作的原则，也接受 CancellationToken 实例。

按照我们刚才对消息处理程序管道组织的描述，消息处理程序需要有一个数据成员，保存指向一个内部处理程序的指针和数据流逻辑，把请求和响应消息从一个处理程序委托给它的内部处理程序。这些附加信息在 DelegatingHandler 类中实现，这个类定义了 InnerHandler 属性，将一个处理程序连接到其内部处理程序。

在 ASP.NET Web API 配置对象模型中，HttpConfiguration.MessageHandlers 集合属性定义了消息处理程序委托的顺序（例如：config.MessageHandlers.Add(new TraceMessage Handler());）。管道中消息处理程序的顺序与 config.MessageHandlers 集合中的顺序一致。

ASP.NET Web API 2.0 引入了 OWIN 模型支持，提供 OWIN 中间件作为消息处理程序的可选方式，实施横切关注点。OWIN 中间件的主要优点是，OWIN 中间件不是专门与 Web

API 绑定的，因而可以与其他 Web 框架（如 ASP.NET MVC 或 SignalR）一起使用。例如，Web API 2.0 中引入的新的安全功能（参见第 15 章），大部分作为 OWIN 中间件实现，并可以在 Web API 之外重用。另一方面，消息处理程序也可以在客户端重用，第 14 章将对此进行介绍。

路由分发

在消息处理程序管道的末端，有两种特殊的处理程序。

- *路由分发器*（routing dispatcher）：由 `HttpRoutingDispatcher` 类实现。
- *控制器分发器*（controller dipatcher）：由 `HttpControllerDispatcher` 类实现。

*路由分发器*处理程序执行下列操作。

- 从消息中获取路由数据（例如，使用 Web 托管时）或者执行之前没有执行的路由解析（例如，使用自托管时）。如果没有找到符合条件的路由，处理程序就产生一个状态码为 `404 Not Found` 的响应消息。
- 使用路由数据，根据匹配的 `IHttpRoute` 来选择转发请求所用的下一个处理程序。

*控制器分发器*处理程序负责执行下列操作。

- 使用路由数据和*控制器选择程序*（controller selector），获得一个*控制器描述*（controller description）。如果没有找到控制器描述，处理程序就返回一个状态码为 `404 Not Found` 的响应消息。
- 获取控制器实例，调用控制器的 `ExecuteAsync` 方法，传入请求消息。
- 处理控制器返回的异常，将其转换为状态码为 `505 Internal Error` 的响应消息。

例如，如果使用示例 4-1 中的 HTTP 请求和默认的路由配置，路由数据就只包含一个带 `controller` 键和 `process` 值的接口。这个路由数据接口是通过匹配请求 URL（http://localhost:50650/api/processes?name=explorer）和路由模板（`/api/{controller}/{id}`）得到的。

默认情况下，*路由分发程序*把请求消息转发给*控制器分发程序*，然后控制器分发程序会调用控制器。但是，我们也可以直接定义一个单路由处理程序（per-route handler），如图 4-4 所示。

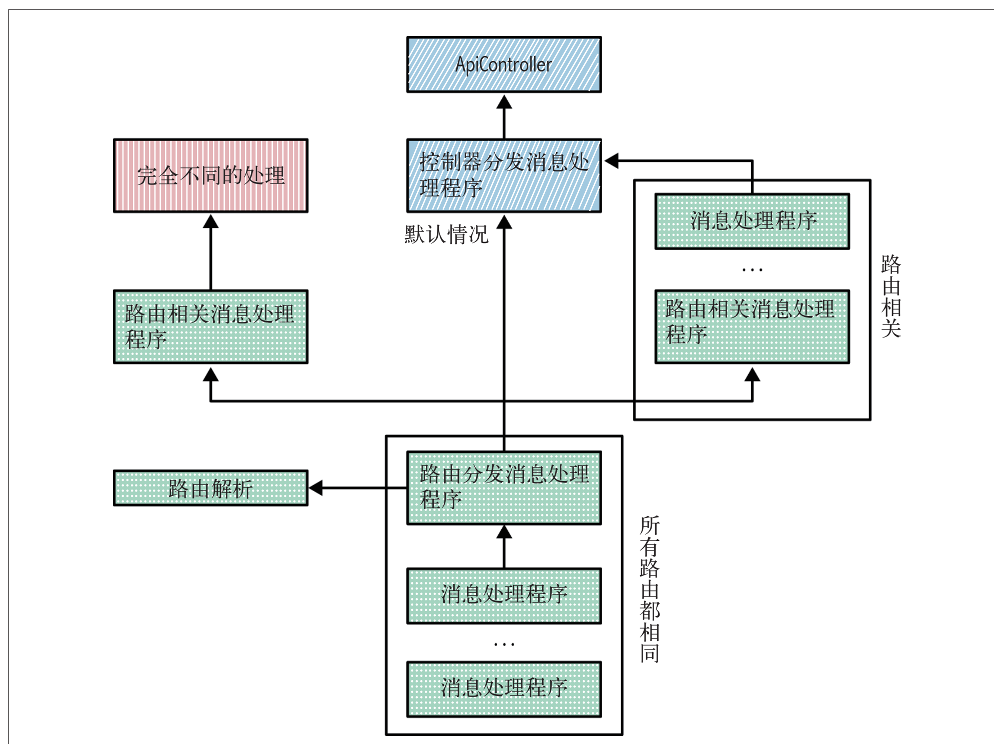


图 4-4：单路由处理程序和路由分发处理程序

对于单路由处理，请求会转发到一个由此路由定义的处理程序，而不是默认的控制器分发程序。使用单路由分发的原因之一，是需要让请求消息通过一个特殊路由的处理程序管道。例如，对不同的路由，通过消息处理程序实现时，会采取不同的身份验证方法。使用单路由分发的另一个原因，是用来替换非 ASP.NET 的 Web API 顶层框架（控制器处理）。

4.3　控制器处理

ASP.NET Web API 处理架构的最后、也是最上一层是控制器处理。这一层负责从底层的管道接收请求信息，转换为对控制器操作方法的调用，并传递需要的方法参数。控制器处理层还负责把操作方法的返回值转换成响应消息，回传给消息处理程序管道。

连接消息处理程序管道和控制器处理层的桥梁是*控制器分发程序*。控制器分发还是一个消息处理程序，主要任务是选择、创建和调用正确的控制器来处理请求。第 12 章将详细讨论控制器分发的过程，介绍所有与这个过程相关的类，展示如何使用可用的扩展点修改默认行为。

ApiController基类

最终处理请求的那个具体控制器，可以直接实现 IHttpController 接口。但是，正如之前一章介绍的，通常的做法是从抽象类 ApiController 进行派生，生成具体的控制器，如图 4-5 所示。

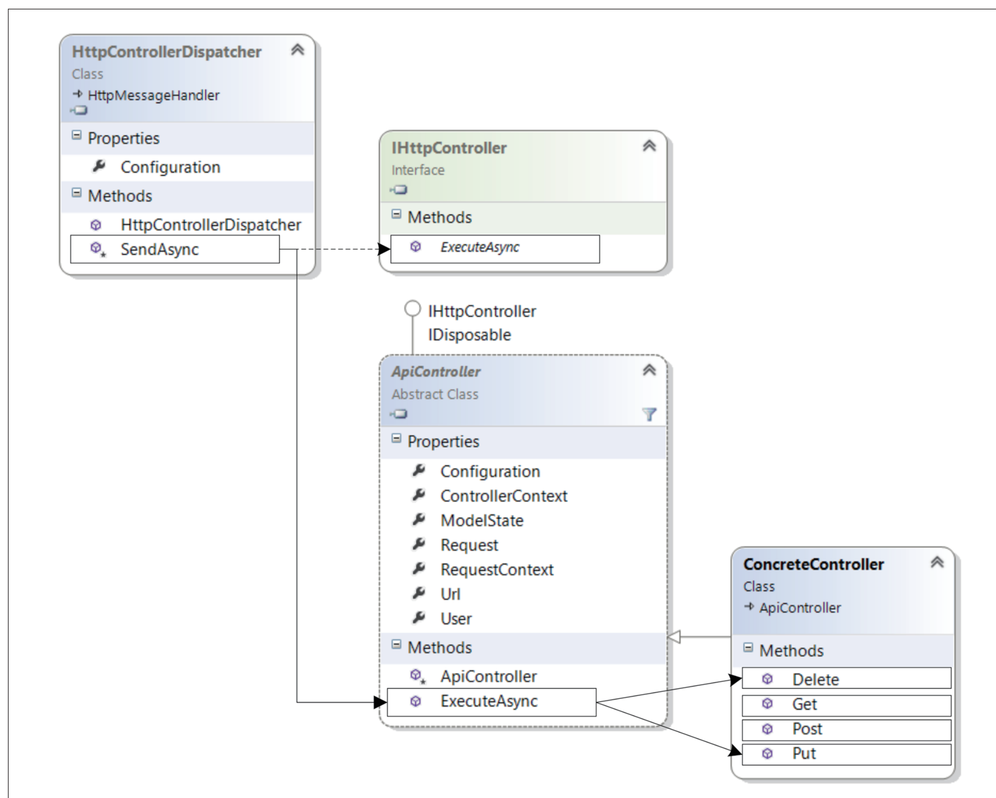


图 4-5：从抽象类 ApiController 派生出具体的控制器类

ApiController.ExecuteAsync 负责根据 HTTP 请求方法（如 GET 或 POST）选择适当的操作，并调用派生的具体控制器上的相关方法。例如，示例 4-1 中的 GET 请求会分发给 ProcessController.Get(string name) 方法。

在选择操作之后，调用相关方法之前，ApiController 类会执行筛选器管道（Filter Pipeline），如图 4-6 所示。每个操作都有自己的管道，具有如下功能：

- 参数绑定；
- 把操作返回值转换为 HttpResponseMessage；
- 身份验证、授权和操作筛选器；
- 异常筛选器。

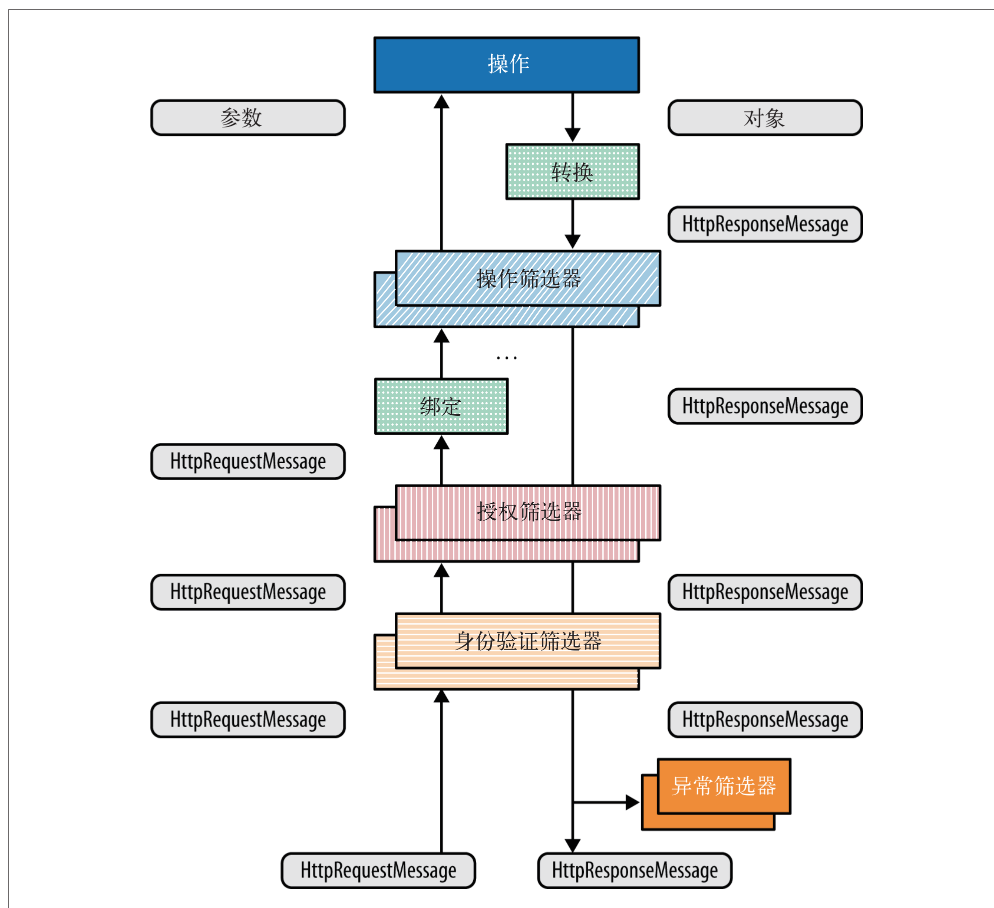


图 4-6：每个操作的筛选器管道、参数绑定和结果转换

1. 参数绑定

参数绑定就是计算操作的参数值，在调用操作的方法时会用到。图 4-7 展示了参数绑定的过程。参数绑定使用来自几个地方的信息，分别是：

- 路由信息（如路由参数）；
- 请求 URI 查询字符串；
- 请求正文；
- 请求标头。

在执行一个操作的管道时，`ApiController.ExecuteAsync` 方法将调用一系列 `HttpParameterBinding` 实例，每个实例与操作方法的一个参数相关。每个 `HttpParameter Binding` 实例会计算一个参数值，把参数值添加到 `HttpActionContext` 实例的 `Action Arguments` 字典中。

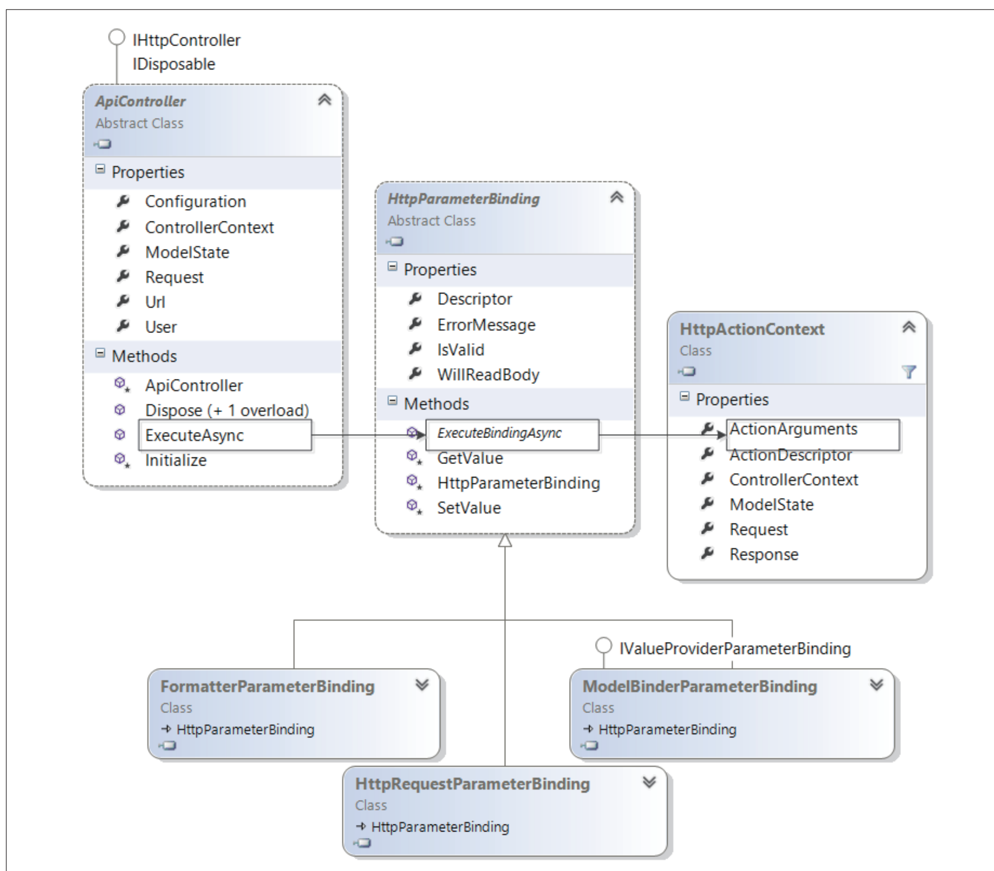


图 4-7：参数绑定

HttpParameterBinding 是一个抽象类，其上派生出多个具体类，每个具体类对应一种参数绑定。例如，FormatterParameterBinding 类使用请求正文内容和一个格式化程序（formatter）获得参数值。

格式化程序是抽象类 MediaTypeFormatter 的扩展类，在 CLR（Common Language Runtime，通用语言运行时）类型和因特网媒体类型（参见 **http://www.iana.org/assignments/media-types/media-types.xhtml**）定义的字节流表示之间进行双向转换。图 4-8 展示了这些格式化程序的功能。

另一种参数绑定是 ModelBinderParameterBinding 类，这个类使用模型绑定（model binder）的概念，采用一种类似 ASP.NET MVC 的方式，从路由数据取得信息。例如，对于示例 4-2 中的操作和示例 4-1 中的 HTTP 请求，GET 方法中的 name 参数会绑定到值 explorer，也就是查询字符串条目中键为 name 的值。第 13 章将更详细地介绍格式化程序、模型绑定和验证。

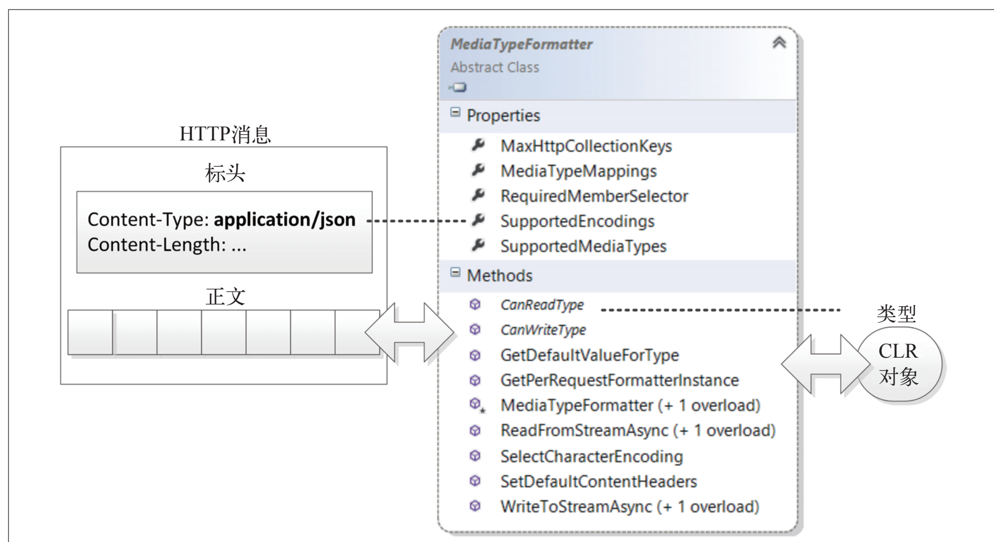


图 4-8：格式化程序以及消息正文与 CLR 对象间的转换

2. 转换为HttpResponseMessage

操作方法的结果可能是任何对象，在操作方法结束之后，结果返回到筛选器管道之前，这个操作结果必须转换为 HttpResponseMessage。如果返回值类型可以赋值给 IHttpActionResult 接口[2]（参见示例 4-3），那么系统就调用这个结果的 ExecuteAsync 方法，将其转换为一个响应信息。IHttpActionResult 接口有好几种实现可以在操作方法的代码中使用，如 OkResult 和 RedirectResult。ApiController 基类中也有几个受保护方法（图 4-5 中没有展示），可以由派生类调用，构建 IHttpActionResult 的实现（如 protected internal virutal OkResult OK()）。

示例 4-3：IHttpActionResult 接口

```
public interface IHttpActionResult
{
    Task<HttpResponseMessage> ExecuteAsync(CancellationToken cancellationToken);
}
```

如果操作方法的返回值不是 IHttpActionResult，那么系统会选用一个实现了示例 4-4 中定义的 IActionResultConverter 接口的外部结果转换程序，生成响应消息。

示例 4-4：结果转换程序　将操作的返回值转换为响应消息

```
public interface IActionResultConverter
{
    HttpResponseMessage Convert(
        HttpControllerContext controllerContext,
        object actionResult);
}
```

注 2：Web API 版本 2.0 中引入了 IHttpActionResult 接口。

对于示例 4-1 中的 HTTP 请求，选中的结果转换程序会试图找到一个能够读取 ProcessCollectionState（即操作方法返回值的类型）的格式化程序，生成 application/json（即请求消息 Accept 标头的值）格式的字节流表示。最终，生成的响应消息如示例 4-5 所示。

示例 4-5：HTTP 响应

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
Content-Type: application/json; charset=utf-8
Expires: -1
Server: Microsoft-IIS/8.0
Date: Thu, 25 Apr 2013 11:50:12 GMT
Content-Length: (...)

{"Processes":[{"Id":2824,"Name":"explorer",
"TotalProcessorTimeInMillis":831656.9311}]}
```

第 13 章将详细讨论格式化程序和内容协商。

3. 筛选器

示例 4-6 中列出的接口定义了身份验证、授权和操作筛选器，这些筛选器和消息处理程序的作用相似，即实现横切关注点（如身份验证、授权和验证）。

示例 4-6：筛选器接口

```
public interface IFilter
{
    bool AllowMultiple { get; }
}

public interface IAuthenticationFilter : IFilter
{
    Task AuthenticateAsync(
        HttpAuthenticationContext context,
        CancellationToken cancellationToken);

    Task ChallengeAsync(
        HttpAuthenticationChallengeContext context,
        CancellationToken cancellationToken);
}

public interface IAuthorizationFilter : IFilter
{
    Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(
        HttpActionContext actionContext,
        CancellationToken cancellationToken,
        Func<Task<HttpResponseMessage>> continuation);
}

public interface IActionFilter : IFilter
{
    Task<HttpResponseMessage> ExecuteActionFilterAsync(
```

```
            HttpActionContext actionContext,
            CancellationToken cancellationToken,
            Func<Task<HttpResponseMessage>> continuation);
    }

    public interface IExceptionFilter : IFilter
    {
        Task ExecuteExceptionFilterAsync(
            HttpActionExecutedContext actionExecutedContext,
            CancellationToken cancellationToken);
    }
```

对于授权和操作筛选器，其管道的组织和消息处理程序管道类似：每个筛选器都有指向管道中下一个筛选器的指针，能够对请求和响应执行预处理和后处理。除了这种执行方式外，筛选器也可以产生一个新响应，立即终止请求，从而取消之后的处理（即调用这个操作）。身份验证筛选器的工作模型稍微有些不同，第 15 章将对此进行介绍。

授权筛选器和操作筛选器之间最主要的区别是，授权筛选器在参数绑定之前执行，而操作筛选器在绑定之后执行。因此，授权筛选器非常适合作为一个扩展点，插入应该尽可能在管道早期执行的操作，例如，验证请求是否得到授权，如果没有得到授权则立刻产生一个返回码为 `401`（`Not Authorized`）的 HTTP 响应消息。另一方面，操作筛选器则适合那些需要访问已绑定参数的操作。

第四种筛选器类型，异常筛选器，只有当筛选器管道返回的 `Task<HttpResponseMessage>` 状态错误时（即发生异常时）才使用。每个异常筛选器都按顺序调用，都有机会创建一个 `HttpResponseMessage`，进行异常处理。回想一下，如果控制器分发程序收到一个未处理的异常，就会返回一个状态码为 `500`（`Internal Server Error`）的 HTTP 响应消息。

我们可以采取多种方式将筛选器关联到控制器或操作：

- 通过属性，与 ASP.NET MVC 支持的方式类似；
- 使用 `HttpConfiguration.Filters` 集合，明确地将筛选器实例注册到配置对象；
- 在配置服务的容器中，注册 `IFilterProvider` 的实现。

`HttpResponseMessage` 实例离开操作管道之后，就由 `ApiController` 返回给控制器分发处理程序，然后在消息处理程序管道中逐步下行，最后由托管层转换成一个本地的 HTTP 响应。

4.4　小结

这本书的第一部分到这里就结束了，这一部分的目标是介绍 ASP.NET Web API，它存在的原因、基本编程模型以及核心处理架构。掌握了这些知识，在这本书的下一部分，我们将把注意力转移到如何以 ASP.NET Web API 为支持平台，设计、实现和使用可演化的 Web API 上。

第二部分

真实世界的API开发





第5章
应用程序

要么演化，要么消亡。

到目前为止，我们讨论了构建 Web API 所需的工具。我们讨论了 HTTP 协议的概念、使用 ASP.NET Web API 的基础知识，以及 Web API 架构的各部协同工作的原理。这些知识都非常重要，却不是这本书的唯一目标。这本书还要讨论如何构建可演化的 Web API。从本章开始，我们就要谈到如何创建一个可以演化数年的 Web API——这段时间相当长，业务关注点和技术都会在此期间发生改变。

我们不会在抽象场景中讨论如何构建可演化的 Web API，而是将实际动手构造一个 API，以演示需要传达的概念。这个 API 所在的应用域应该是每个开发者都熟悉的，而且足够现实，可以在真实世界场景中使用。

在深入这个应用域的细节之前，我们必须确定自己真的做好准备，为获得可演化性付出努力。可演化性并不容易实现。为了获得可演化性，在适当的时候，我们会使用 REST 架构风格的约束。关键是要认识到，我们并不是要努力创建一个“RESTful API”。REST 并不是目标，而是实现目标的手段。在某些情况下，我们很可能会选择违反 REST 的某些约束。一旦理解了一个架构约束的价值和需要为之付出的代价，你就能够做出明智的选择，决定这个约束是否与你的目标一致。可演化性才是我们的目标。

在开始设计 Web API 之前，我们需要定义应用域的组成部分。如今，人们试图构建分布式系统时经常会忽略这个定义过程。如果你构建的是一个经典的基于浏览器的应用程序，解决方法的架构基本上已经定义好了。HTTP、HTML 页面、CSS 和 JavaScript 都是这个解决方案的组件。但是，如果要构建 Web API，你就不再受限于 Web 浏览器所提供的选择。

Web 浏览器可能只是这个分布式系统中的诸多客户端组件之一。

在定义了应用域的组成部分之后，第 7 章将介绍一个 API 示例，把这些组件集成为 Web API。这些组件也是在第 8 章中构建客户端程序所需的关键信息。通过定义独立于 API、可重用的组件，我们可以构建这样的客户端：可以与这些组件进行交互，而无需知道 Web API 的具体类型。通过使用超媒体，客户端可以发现 API 的具体类型，并在 API 演化过程中适应其变化。

5.1 为什么要可演化

到底什么是可演化的 API 呢？在一定程度上，演化只是变化的另一种华丽的说辞而已。但是，演化意味着一组连续的微小变化，在经过多次迭代后，最终的解决方案可能会看起来和原来完全不同。

在 Web API 生命周期中，可能会需要进行如下的演化。

- 个体资源包含的信息可能需要增加或者减少。
- 单个信息的表示可能发生变化。名字或者数据类型可能改变。
- 资源之间的关系可能会增加或删除，或者关系的基数发生改变。
- API 中可能增加全新的资源，用于表示新的信息。
- 资源可能需要新的表示，以支持不同类型的客户端。
- API 可能创建新资源，提供更细粒度或更粗粒度的信息访问。
- API 支持的处理流程可能发生变化。

多年以来，软件开发行业在变更管理方面，一直试图遵循较为传统的工程实践。传统经验表明，在产品开发周期的早期进行变更，要比开发晚期进行变更的代价小得多。常见的解决方案是进行严格的变更管理，通过全面的前期计划和设计工作，尽量减少变更。近年来，敏捷软件开发实践的兴起为我们指出了另一种对待变更的方式：把变更看做软件开发过程的一部分。在小的迭代中接受变更，软件系统可以借此进行演化，以满足用户的需求。我们有时会看到网站在一天之内多次发布软件的新版本。

但是，Web API 允许外部团队开发自己的软件使用其服务，因此，对于 Web API 的变更管理，我们又回到了老习惯。我们的想法通常是：如果修改 API 导致软件不能运行，客户会不高兴，因此我们需要对 API 做出正确的计划和设计，避免将来进行修改。我们经常听说 API 开发者发布了 2.0 版本的 API，导致客户端应用需要进行许多修订，才能移植到新版本。

> **SOAP 至少是诚实的**
>
> 使用基于 SOAP 的解决方案虽然存在诸多问题，但是 SOAP 解决方案至少在变更的控制和管理上是“诚实”的。SOAP 明确地要求，在 API 提供者和使用者之间定义准确的协议，使用瀑布式的集成方式，做好前期工作，从而避免变革。在今天，很多人采取与 SOAP 解决方案完全不同的做法，他们遵循 REST 风格开发系统，希望 REST 风格系统具有某些神奇的属性，能够使变更管理变得更容易。遗憾的是，这些人只是把问题推后了而已。

5.1.1 演化的障碍

有几个因素会影响变更处理的难易程度。最大的因素之一是，谁会受到变更的影响。API 的使用者与开发者通常是不同的团队。而且，使用者团队经常和开发团队隶属不同的公司。实际上，API 可能会有来自多个公司的多个使用者。实施一个破坏性的变更会导致很多顾客不愉快。

即便是同一个团队管理使用者和生产者两个应用，也可能有约束条件导致客户端和服务器无法同步进行部署。如果客户端代码要部署到许多客户端，就很难同步客户端和服务器的更新；如果客户端安装在锁定的机器上，更新可能会变得更加复杂。有时，强制客户端进行更新会导致糟糕的用户体验。如果用户想要使用一个客户端程序，在使用前却必须安装一个更新，用户也许会觉得很扫兴。大部分常见的自动更新的客户端软件应用可以在运行的同时，后台下载新版本，然后下一次重启时进行更新。要使用这种更新方法，服务器软件需要能够至少在短期内，继续支持客户端的旧版本。

修改软件的挑战之一是，有些变更非常简单，影响很小，而有些变更会严重影响系统的很多部分。关键在于区分你要进行的是哪种变更。在理想情况下，我们应该能够将软件分块，将改动影响很小的部分与可能产生很大影响的部分分开。举个简单的例子，让我们对比 HTML 规范的修改和在网站添加一个新页面两种变更。修改 HTML 规范可能导致每个人都要更新 Web 浏览器，影响范围巨大，而网站添加一个新页面则不会对 Web 浏览器产生任何影响。Web 架构的设计特意做出了这种区分，使得网站（服务器）可以独立进行演化，而用户无需不断更新 Web 浏览器（客户端）。

变更需求可以进行管理。变更需求的管理需要控制和协调。我们能够采取的任何辅助长期变更的努力，都很容易得到数倍的回报。但是，构建可演化的系统并不容易，我们必须付出代价。

5.1.2 代价是什么

为了使 API 可演化，REST 的约束不允许客户端应用程序做出某些假设。我们不允许客

户端事先知道服务器上可用的资源。客户端必须在运行时，使用一个入口点 URL 来发现资源。

客户端发现了资源的 URL 之后，也不能对可能返回的资源表示做出任何假设。客户端必须使用响应中返回的元数据，判断返回信息的类型。

在这些限制下，相比传统的客户端 / 服务器模式中的客户端，我们构建的客户端要动态得多。客户端必须进行“功能检测”，以决定哪些功能可用，还必须根据返回的响应做出灵活处理。

让我们看一个简单的例子：返回一个订单的服务器 API。假设客户端能够显示纯文本、HTML 和 PDF 文档，服务器不需要预先定义返回何种格式。也许，在软件的第一个版本中，API 返回一个简单的纯文本订单。服务器的新版本可能实现了返回 HTML 订单的功能。只要存在一个机制，客户端可以使用这个机制解析响应，识别返回类型，客户端就可以继续工作，无需改变。这种包含标识消息内容的元数据的消息，称为*自描述性消息*（self-descriptive messaging）。

此外，定义 API 规范时应该说明必需的最少的细节，而不是所有细节。爱因斯坦说：“所有事情都应该尽量简单，但是不能过于简单。”过度地限定 API 会增加变更，影响系统较大部分的可能性。

过度指定的例子有：

- 当数据顺序对消息语义无关紧要时，要求数据按某个顺序排列；
- 当数据只在特定情况需要时，认为数据是必需的；
- 限制交互必须使用平台定义类型的特定序列化方法。

下面的两个小故事描述了现实生活中的场景，说明过度指定如何对结果产生负面的影响。

婚礼摄影师

在这个真实生活场景中，你会看到如何撰写协议，以适应变化。请比较下面两份由新娘提供给婚礼摄影师的协议条款。

(1) 我们希望拍摄如下照片：

- 新娘在教堂内外；
- 新郎和新娘在教堂台阶上；
- 新郎和新娘在教堂门前的大橡树旁；
- 新娘和伴娘在池塘前；
- 新郎和伴郎在礼车旁。

(2) 我们希望拍摄一些可以赠予家人及亲友的照片。我们也希望拍摄其他更为私人的照片，用于装饰家居。我们希望一些照片是黑白的，以配合客厅的装饰。比起室内照片，我们更喜欢室外照片。

第二份协议与第一份的不同之处在于，第一份描述非常详细，但是没有解释条款的意图。第二份更加灵活，同时也保证客户的需求得到满足。第二份协议留给摄影师更多的创作自由，如果婚礼当天摄影师发现大橡树已经被砍掉，或者礼车停在一条繁忙的街道上，无法避开背景中的车辆时，摄影师可以灵活处理。

下面这个例子中，过度指定的结果不是简单的压抑创造力和产生低质量的产品，而是完全没有满足原本的需求。

遗嘱的意图

变化是不可避免的，如果今天做出过于精确的决定，将来环境变化时可能会适得其反。一位有四个孩子的母亲正在准备遗嘱。这位母亲深爱她的每一个孩子，因为孩子们的经济状况都不是特别好，她希望在自己去世后帮助孩子们。但是 Johnny 沉迷赌博，母亲不想把钱浪费在他身上。母亲决定在遗嘱中不包括 Johnny，把钱平均分给其他三个孩子。不幸的是，母亲突发中风陷入昏迷，几年之后终于去世。在母亲昏迷的这段时间里，Johnny 不再赌博，重新振作。Billy 买彩票中了 1 千万，逐渐与家人疏远。Jimmy 失去工作，生活十分困难。母亲立下的遗嘱不再反映她的本意。她本想尽量用金钱帮助自己的孩子们，但是 Billy 不再需要金钱，Johnny 却很需要资金使生活回到正轨。可惜这份遗嘱在钱财如何分配上定义得非常严格。如果这位母亲选择灵活的措辞，让遗嘱执行人能够满足她立定遗嘱的最初意图，结果本可以有很大的不同。

我们经常在软件项目管理中看到类似的情况：客户和业务分析师试图用需求来精确定义他们设想的解决方案，而不是说明他们的真实意图，让专业的软件人员发挥所长。在构建分布式系统时，如果希望系统能够长期运行，我们使用的协议必须表达业务伙伴的意图。

将运行时发现、自描述性消息和反应性客户端结合在一起，其概念并不容易掌握，也不易实施，但这是可演化系统的核心组件，带来的灵活性要远远超过实施的代价。

5.1.3　为什么不创建新版本

处理 API 中的破坏性变更，传统的做法是使用版本的概念。

从开发可演化系统的角度，我们可以把创建新版本当作最后一个办法，承认演化的失败。给初始 API 冠上版本号 v1，等于是声明，你已经知道这个 API 无法演化，需要在 v2 版本

引入破坏性的变更。但是，有时我们的确会犯错误，如果试过别的方法都不成功，那么只能选择创建新版本。

这本书中讨论的技术，是为了帮助你避免为 API 创建新版本。但是，如果确实需要为 API 的某些部分创建新版本，你可以试着缩小演化失败的范围。

请不要误会，我们不是说你必须在前期做大规模的设计，一开始就把事情都做对。我们是想鼓励一种极简化的态度：不要指定不需要指定的东西；不要创建不需要的资源。可演化的 API 设计就是为了适应变化，在发现必须添加的新东西时，能够以最小的代价完成所需的改变。

版本变更，包括把一个标识符与 API 快照或 API 的某部分联系起来。如果 API 发生变化，就需要使用新的标识符。客户端和服务器可以使用版本号进行协调，客户端用版本号判断是否能与服务器进行通信。一些概念（如语义版本）可以用于区分破坏性变更和非破坏性变更。在可演化系统内可以使用几种版本创建方式，每种方式对系统的影响程度不同。

我们可以在如下位置进行版本变更：

- 有效载荷内（如 XML 和 HTML）；
- 有效载荷类型（如 `application/vnd.acme.foo.v2+xml`）；
- URL 开头（如 /v2/api/foo/bar）；
- URL 结尾（如 /api/foo/bar.v2）。

什么样的变更是破坏性的？

如果我们可以把 API 变更划分为破坏性和非破坏性两种，生活会变得很轻松。遗憾的是，具体的上下文对我们的判断有很大的影响。一个破坏性变更是会破坏协议的变更，但是如果没有严格简单的规定说明 API 协议是什么，我们还是无法做出判断。在 REST 架构风格中，协议是由媒体类型和链接关系定义的，因此我们认为，不影响媒体类型和链接关系的变更是非破坏性变更，而影响这两种规范的变更可能是破坏性的，也可能不是破坏性的。

1. 基于有效载荷的版本变更

Web 架构最重要的特征之一是，Web 将有效载荷格式的概念提升为第一类的架构概念。在 RPC 世界中，参数仅仅是过程签名的产物，不能独立使用。HTML 是基于有效载荷进行版本变更的一个例子。HTML 在很大程度上是一个自包含的规范，描述一个文档的结构。HTML 规范多年来经过了巨大的演化。基于 HTML 的 Web 使用的不是基于 URI 的或媒体类型的版本变更。但是，HTML 文档的确包含元数据，以辅助解析器解释文档的含义。这种版本变更的方式可以限制版本变更对媒体类型解析器代码的影响。我们可以创建支持不

同版本的数据传输格式的解析器，更容易支持旧版本的文档格式。但是要确保新的文档格式不会导致旧的解析器失败，还是一个挑战。

2. 媒体类型版本变更

近年来，使用媒体类型标识符版本的做法日渐流行。这种方法的一个好处是，用户代理可以使用 `Accpet` 标头声明自己支持媒体类型的哪个版本。如果客户端处理得当，你可以在媒体类型中引入破坏性变更，而现有客户端依然可以工作。现有的客户端会继续请求旧版本的媒体类型，新客户端可以使用新版本的媒体类型。

> **不透明标识符**
>
> 从 HTTP 的角度来看，媒体类型的版本变化根本就算不上是真正的版本变更，只是创建了一个新的媒体类型而已。对于 HTTP，媒体类型标识符是不透明的字符串，不能从这个字符串中推导出含义。因此，`application/vnd.acme.foo` 和 `appliction/vnd.acme.foo.v2` 一样，都与 `text/plain` 没什么关系。这些标识符字符串各不相同，因此代表的媒体类型不同。至于这两个版本的媒体类型的解析代码可能相似度达到 99%，不过是实现上的细节罢了。

使用媒体类型创建版本有一个缺点，这种做法加剧了服务器被动协商中存在的一个问题。一个服务有可能为了提供各种内容而使用很多不同的媒体类型。内容协商要求客户端在每个请求中都声明自己能够显示的所有媒体类型，给每个请求增加了不小的开销。在媒体类型中加入版本信息会使这个问题更加复杂。如果一个客户端支持某个特定媒体类型的 v1、v2 和 v3 版本，那么，为了避免服务器只支持旧的版本，我们需要在每个 `Accept` 标头中都加入这三个版本吗？有些用户代理开始采取一种做法，根据需要访问的链接关系，在 `Accept` 标头中只发送一部分媒体类型。这样可以缩小 `Accept` 标头的大小，但是用户代理必须能够把链接关系对应到适当的媒体类型，又增加了额外的复杂度。

3. URL版本变更

URL 中的版本变更可能是在公共 API 中最常见的做法。更准确地说，公共 API 经常在 URL 的第一段加入一个版本号，如 http://example.org/v2/customers/34。这种做法也是 REST 追随者批评得最厉害的。反对者认为，如果在 URL 中加入一个新的版本号，那么你就隐含地创建了一组带有新版本号的资源副本，而这些资源大部分可能并没有发生改变。如果 URL 之前已经发布，那么这些 URL 将会指向资源的旧版本，而不是新版本。问题在于，有时候这种行为是我们希望得到的，而有时候不是。如果客户端有使用新版本资源的能力，那么客户端会希望使用新版本，而不是之前保存的旧版本。如果新版本的资源实际上和旧版本一样，就产生了一个新问题：两个不同的 URL 指向同一个资源。如果你想利用 HTTP 的缓存功能，指向同一资源的多个 URL 会带来许多弊端，最终导致缓存中保存了同一资源的多个副本，缓存失效操作变得效率低下。

URL 版本变更的另一种做法是在 URL 末尾附近加入版本号，如 http://example.org/customers/v2/34。采用这种方法，API 内的单个资源可以独立创建版本，消除了单个资源对应多个 URL 的问题。对于需要构造 URL 的客户端来说，这种带版本信息的 URL 构造起来比较困难，不能简单地替换所有请求 URL 的前段路径。超媒体驱动的客户端甚至无法使用这种版本变更方法，因为 URI 对超媒体驱动的客户端是不透明的，新的版本信息必需通过带版本的链接关系来获得。

API 的版本变更非常困难，很容易出错。我们还是应该尽一切可能避免进行版本变更，这种努力会带来很多长期的益处。

5.1.4　付诸实践

本章到现在为止，我们已经大致讨论了开发可演化应用程序的利弊。在开发可演化的分布式应用过程中，很多时候你需要做出选择，而正确的答案经常是："视情况而定。"我们不可能在有限的篇幅内，讨论每一种可能性和每一种结果。但是，展示对于某些情况做出的某些选择，还是有价值的。在本章余下的部分，我们将关注一个具体的应用程序，这个应用具有常见的问题，我们将讨论各种选择，做出决定。我们在随后的章节中，为每种情况做出的选择未必最佳，但是这些选择可以作为示例，告诉你在构建可演化 API 时将面对的各种选择和需要做出的决定。

5.2　应用程序目标

为演示用的应用程序示例挑选应用域总是特别困难。这个域应该是真实的，但是不能纠缠于过多的专业细节；这个域应该足够复杂，可以提供各种场景，但是不能过大，以免架构原则淹没在实现细节中。为了免除你学习专业领域知识的麻烦，我们选择了一个软件开发者都很熟悉的应用域：问题跟踪。我们都见过各种问题跟踪系统的实现，既有客户端的，也有服务器的。问题跟踪系统主要关注的是不同团队成员之间的沟通和信息共享，因此很自然地是分布式的。问题的生命周期有很多的用例。问题有很多不同的类型，各有不同的状态集。问题有很多相关的各种元数据。

5.2.1　目标

对于认定属于问题跟踪应用域的信息，我们要为其定义类型。我们希望能够：

- 定义表示一个问题所需的最小的一组信息；
- 定义通常与问题相关的一组核心信息；
- 确定系统中信息之间的关系；
- 定义问题的生命周期；
- 定义与问题相关的行为；

- 对于可以在问题上执行的聚合、过滤和统计类型进行分类。

我们并不会试图完备地定义最终的，无所不包的问题结构。我们的目标不是定义问题跟踪的应用域中每个可能场景的功能，而是要定义一组通用的信息和术语，用于把信息传达给希望构建某种问题跟踪应用程序的人，而不局限于这些应用程序的范围。我们定义的内容将会发生演化。

如果你对试图解决同一领域问题的不同应用程序进行比较，经常会发现这些应用解决同一问题的方式略有不同。我们要确定什么情况下不同的选择是无关紧要的，直接挑选一个，或者两个选项都启用。当我们最终把这个域提炼成媒体类型规范、链接关系和语义档案时，就应该得到一个通用的基础，既提供一定程度的交互性，又不必局限单个应用程序于提供单一功能。

5.2.2 机会

问题跟踪应用域是时候需要改进了。问题跟踪有不少商业应用和开源的应用，但是这些应用都使用专用格式进行服务器和客户端组件间的交互。问题数据锁定在专用数据存储库中，而这些数据存储库与创建这些数据的应用紧密相关。要访问一个特定的问题数据存储库，我们只能使用专为这个存储库设计的客户端工具。为什么我不能使用我选择的问题管理客户端，同时管理在 Bitbucket 和 GitHub 存储库中的工作事项呢？为什么要有多个存储库呢？从来没有人想编写专门用于 Apache 或 IIS 的 Web 浏览器，那么在其他使用分布式数据的应用领域，为什么我们要坚持把客户端和服务器应用捆绑在一起？

遗憾的是，通过重用标准媒体类型，进行 Web 架构的重构和重用，似乎并不符合商业组织的利益。通常，只有在开源项目推动事情发展之后，商业组织才会注意并且认识到，集成和互操作性实际上也会给商业软件带来巨大的好处。

5.3 信息模型

在开始定义媒体类型、链接关系或语义档案之前，我们必须对在 Web 上进行通信所需的语义有一个更为清晰的理解。

在根本上，用一个简短的字符串文本就可以描述一个问题，例如“在屏幕 *Y* 点击按钮 *X* 时应用程序会失败”。此外，人们通常希望问题定义包含更详细的描述。

请看下面这个极为简单的问题定义：

```
Issue
  Title
  Description (Optional)
```

如果有人要了解并解决这个问题，这个定义可能就足够了。虽然这个问题没有包含创建者和创建时间信息，但是这些信息可以在发出创建这个问题的请求时，从可用的环境信息中获取。这个极简的问题表示可以作为一个容易实现的起点，之后再进行演化。使用这种最简单的定义，我们可以很快构建出能够运行的程序，进行演示，而且也可用于比较简易的客户端，如电话。人们可以在问题创建之后，使用功能更强的客户端添加更多的信息。

5.3.1 子域

在匆忙开始实现这个极简的表示之前，我们还需要更好地了解可能与问题相关的各种数据。为了更好地组织内容，我把这些信息分为四个子域：描述信息、分类信息、状态信息和历史信息。

1. 描述信息

描述信息子域包括我们已经讨论过的信息（如标题和描述），还有环境信息，如软件版本、主机操作系统、硬件配置、重现步骤以及屏幕截图。任何与问题产生环境相关的详细信息都可以归在这个子域。描述信息的一个重要特征在于，它主要只是供人工读取的。描述信息不会影响问题的工作流或者对任何算法产生影响。

2. 分类信息

分类信息主要用于将问题进行有意义的分组。问题可以带有一些属性，这些属性值的范围预先已经定义，用于问题的分类处理，例如：优先级、严重性、软件模块、应用领域以及问题类型（缺陷、功能等）。分类信息用于问题的搜索、过滤和分组，经常用于决定应用程序的工作流。通常，分类信息在问题生命周期的早期指定，除非信息指定错误，否则不会发生改变。

3. 状态信息

问题通常有一组属性，用于定义当前状态，例如：当前工作流状态、问题的所有者、剩余时间和完成进度。在问题的生命周期中，状态信息会多次改变。状态信息也可以用做分类属性。问题的当前状态也可能带有文本注释。

4. 历史信息

历史信息通常记录之前某个时间点问题的状态。历史信息一般对问题的处理并不重要，但是可以用于分析过去的问题，或者调查某个问题的历史记录。

5.3.2 相关资源

我们前面提到的所有信息，都可以用两种方式之一表示——本地数据类型的简单序列化（如字符串、日期和布尔值），或者代表另一个资源的标识符。例如：要标识相关的人员，我们可能使用 `IssueFoundBy` 和 `IssueResolvedby`。

我们可以简单地使用一个字符串标识相关人员，但是使用资源标识符要有价值得多，因为问题跟踪系统的用户很可能会定义为资源。资源标识符自然会使用 URL。如果使用 URL，客户端软件就有机会对 URL 进行非关联化，获得相关人员的更多信息。问题属性和人员属性中的信息有着极为不同的生存期，因此适合使用不同的资源。数据的持久性不同，就可能使用不同的缓存策略。

在人们浏览问题时，我们可能不想在其面前展示 URL。要解决这个问题，我们可以使用链接。通常，我们不会在表示中直接嵌入 URL，因为这个 URL 经常有相关的其他元数据。`Link` 是一个带有相关元数据的 URL，其中一个标准元数据是 `Title` 属性。`Title` 属性提供 URL 的一个人工读取的版本。使用链接，我们可以获得两方面的好处：一个嵌入的、可人工读取的描述；以及一个 URL，指向包含了相关人员的附加信息的唯一资源。

下面是一个相关资源的示例：

```
<resource>
    <Title>App blows up</Title>
    <Description>Pressing three buttons at once causes crash</Description>
        <links>
                <Link   rel="IssueFoundBy"
                                title="Found by"
                                href="http://example.org/api/user/bob"/>
        </links>
</resource>
```

5.3.3 属性组

有时我们需要把属性组织在一起，使表示更容易理解。当一组属性需要作为一个整体处理时，使用属性组可以简化客户端代码。如果用户代理不想处理相关的属性，我们就可以忽略整个属性组。我们还可以使用属性组引入对强制信息的条件需求。例如，如果使用组 X，就必须在组中包含属性 Y。使用这种方法，我们可以支持一个最简的表示，同时还能确保，如果表示包含问题的某个方面信息，那么必须提供关键信息。举一个具体的例子，如果你要提供一个问题在过去某个时间的历史记录，就必须提供日期和时间属性。

但是，定义属性组并在媒体类型中使用这些组，会带来一个风险。如果你发现一个属性分组错误，要将它移动到一个新的组，可能会导致破坏性的变更。因此在使用属性组时必须非常谨慎。

下面是一个属性组的示例：

```
<resource>
    <Title>App blows up</Title>
    <Environment>
        <OperatingSystem>Windows ME</OperatingSystem>
        <AvailableRAM>284MB</AvailableRAM>
```

```
        <AvailableDiskSpace>1.2GB</AvailableDiskSpace>
    </Environment>
</resource>
```

5.3.4 属性组的集合

当你要在一个表示内表示多个同类的属性时，可以使用属性组集合。问题可能带有附加文档。这些文档很可能会表示为链接，但是这些文档可能还有附加的属性，这些属性可以放在一个属性组里。采用这种做法，这些文档的多个属性组可以包含在一个资源表示内，同时还可以维护文档及其相关属性的关系。

下面是一个属性组集合的示例：

```
<resource>
     <Title>App blows up</Title>
     <Documents>
         <Document>
                 <Name>ScreenShot.jpg</Name>
                 <LastUpdated>2013-11-03 10:15AM</LastUpdated>
                 <Location>/documentrepository/123233</Location>
         </Document>
         <Document>
                 <Name>StepsToReproduce.txt</Name>
                 <LastUpdated>2013-11-03 10:22AM</LastUpdated>
                 <Location>/documentrepository/123234</Location>
         </Document>
     </Documents>
 </resource>
```

5.3.5 信息模型与媒体类型

到这里，我们已经介绍了问题跟踪应用域的信息模型。我们还抽象地讨论了这些不同的信息如何表示、组织和关联（参见图 5-1）。我避开了对具体格式（如 XML 和 JSON）的讨论，因为信息模型的定义是独立于具体的表示语法的，理解这一点非常重要。至于如何把概念模型具体映射到实际媒体类型的语法及其特定格式，在下一章讨论媒体类型时，我们将会解决这个问题。

关于这个信息模型的可重用性，我们需要考虑几件事情。虽然这个模型列出的功能不少，但大部分都是可选的，因此我们可以在最简单和最复杂的场景都使用同一个模型。但是，为了实现互操作性，我们必须进行明确的定义，给出具体的属性名。幸运的是，我们定义的只是一个接口规范。应用程序在存储数据时，并不需要使用我们定义的属性名，只要能够向用户准确传达数据的语义就可以。

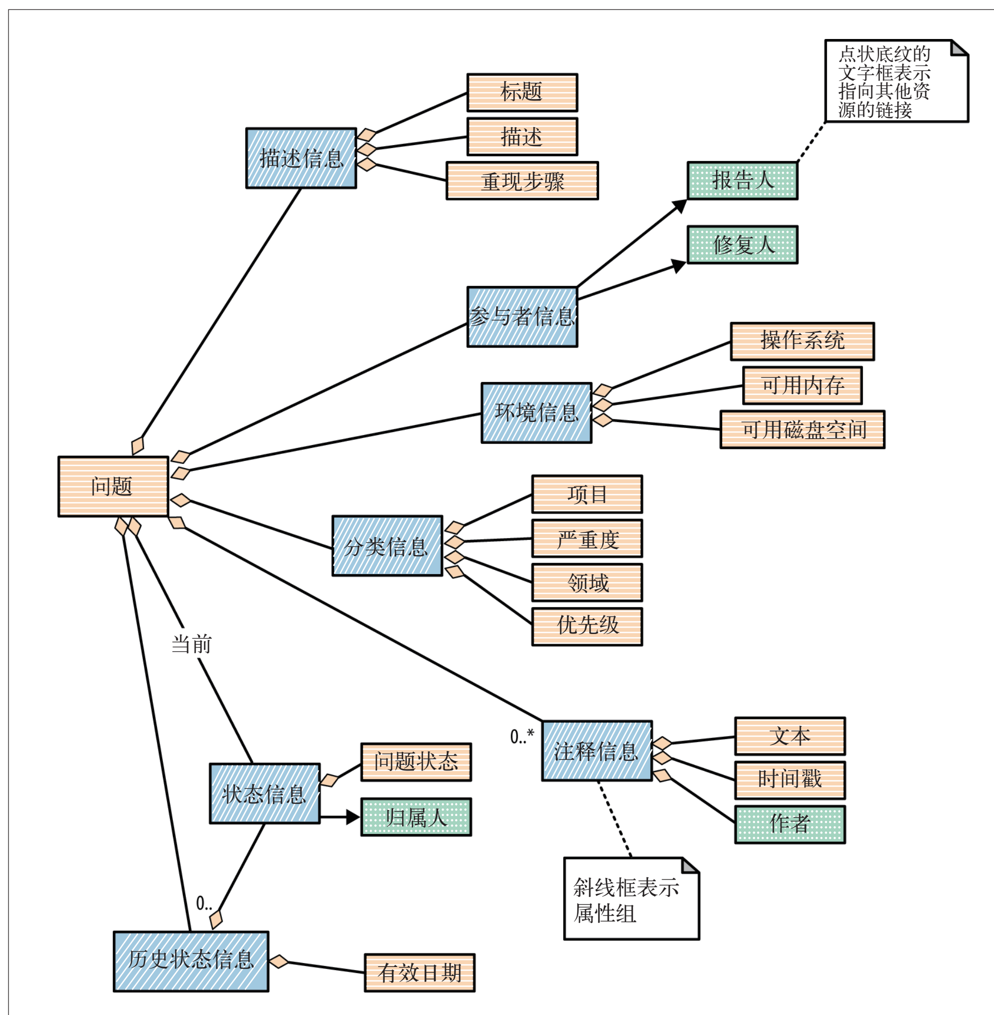


图 5-1：信息模型

在定义媒体类型时，我们必须考虑，如果一个应用程序要包含我们当前的信息模型不支持的语义时，该如何处理。可扩展性是一项重要的目标，但是，构建可以互操作的可扩展性已经超出了我们讨论的范围，因此这个信息模型中对此不做支持。当然，我们并不反对媒体类型定义自己的可扩展性选项，让特殊的客户端和服务器处理扩展数据。

5.3.6 问题集合

除了单个问题的表示，我们的应用程序可能还需要能够表示一组问题。问题集合的表示最有可能在一些查询请求返回的结果中用到。在下一章，我们将介绍两种方法，一种是构建

新的媒体类型用于问题集合的表示，另一种是重用现有媒体类型的列表功能，并讨论二者各自的优点，同时还会介绍现存的各种“混合”做法。

5.4 资源模型

构建 Web API 时，应用架构中需要考虑的另一个主要方面是公布的资源。我并不想预先定义好问题跟踪 API 必须使用的资源。可演化的 API 和 RPC/SOAP API 最主要的区别之一就是，可演化的 API 的接口定义不包括可用的资源。我们预期由客户端发现资源，客户端的功能受限于它能发现的资源。

我想要讨论的是 API 必须公布的资源类型，以帮助我们探索客户端需要支持的媒体类型。从最小的一组资源开始总是没错的。系统中的资源应该能够容易、快速地创建，在我们获得了用户使用服务的真实经验后，就可以很容易地添加新的资源，以满足更多的需求。

5.4.1 根资源

每个可演化的 Web API 都需要一个根资源（root resource）。没有根资源，客户端就无法开始发现资源的过程。根资源的 URL 是唯一不能改动的。这个资源主要会包括指向应用程序内其他资源的一组链接。这些链接有些可能指向搜索资源。

5.4.2 搜索资源

搜索资源（search resource）的一个典型例子是一个 HTML 表单，表单上有一个输入框，使用输入的值作为查询字符串进行 `GET` 请求。搜索资源有可能比这个例子复杂得多，有时能够完全用 URI 模板替代。搜索资源通常会包含一个链接，返回某些资源集合。

5.4.3 集合资源

集合资源（collection reosource）返回的表示通常包含一列属性组，每个属性组经常会包含一个链接，这个链接指向属性组中信息代表的那个资源。

一个问题跟踪应用程序有可能会预先定义很多集合资源，例如：

- 未解决问题；
- 已关闭问题；
- 用户列表；
- 项目列表。

通常，Web API 可以使用搜索参数，生成筛选结果集合，以此定义大部分的集合资源。使用资源这个词时，我们要理解资源这个概念和生成资源表示的底层实现之间的区别，这

一点很重要。如果我创建一个 `IssueController`，为客户端应用搜索问题子集，那么 /issues?foundBy='Bob' 和 /issues?foundBy='Bill' 这两个 URL 是两个不同的资源，虽然生成这两个资源的代码可能是完全一样的。这两个资源是同一概念的不同实例，就我所知，没有一个常用的词可以描述这种资源的共同特征。从这里开始，我会使用*资源类*（resource class）来指代这种情况。

5.4.4 个体资源

通过问题跟踪 Web API 获取的大部分信息都可以通过个体资源（item resource）来传递。个体资源提供一个表示，包含信息模型中某个概念的单个实例的一些或全部信息。我们可能需要支持不同详细程度的表示。例如：一些客户端可能只需要问题的描述性属性，而其他客户端可能需要问题的所有可编辑属性。

客户端在特定的情况下需要的信息子集可能会各不相同。因此在任何 `issue` 相关的媒体类型定义中，对于包含或不包含哪些信息，我们的定义是非常灵活的。一个问题资源的表示不包含历史信息并不意味着这些历史信息不存在。如果我们基于域对象的序列化来生成资源的表示，就无法解决这个问题。一个对象只有一个类定义，因此无法根据上下文来选择序列化这个对象哪些部分。

在考虑资源中应该包含哪些属性时，我们重点需要考虑几个因素。使用大而全的资源表示，可以减少来回交互的次数。但是，在只需要少量数据时，使用大的资源表示就会浪费带宽和处理时间。另外，大的资源表示更有可能包含不同水平的易变性。如果在一个资源中既包含*描述性*属性，又包含*状态*属性，由于状态信息经常发生改变，我们无法随心所欲地长时间缓存资源表示。一个实用的技巧是，参照数据的易变性，把数据组织在不同的资源中。把一个概念分成多个资源的缺点是，使用 `PUT` 方法进行原子更新操作会更加困难，而且会使缓存失效机制变得更为复杂。

使用更多更小的资源，意味着我们要管理更多的链接和表示，但是也意味着有更多重用的机会。

要决定资源定义的最佳粒度，没有公式化的方法可用。关键是要考虑具体的使用场景，以及我们刚才讨论的各种因素，选择对具体场景最合适的资源大小。

图 5-2 展示了一个可能用于问题跟踪服务的资源模型。服务会把每个资源或资源类发布在一个特定的 URL。我们没有展示具体的 URL，因为这些 URL 与设计无关，和客户端也没有关系。

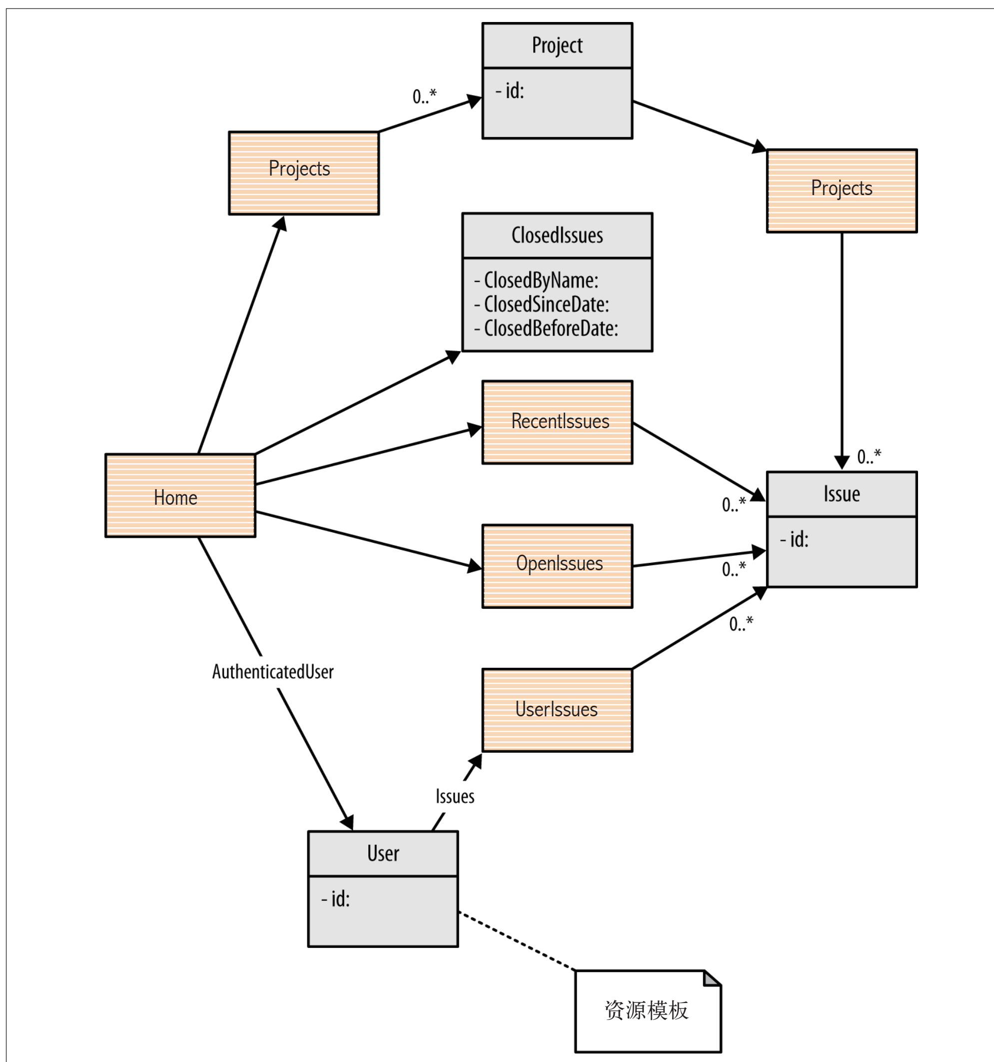


图 5-2：资源模型

很多时候，开发者在构建 API 时会试图“按 URL 设计”（design by URL）。但使用这种方法有若干问题。按 URL 设计会使人们把应用程序定义成层级的数据结构，而不是应该建模的应用程序工作流 / 状态机。所选实现框架在解析、处理和分发 URL 能力上受到限制，会进而约束系统的设计。按 URL 设计还会鼓励开发者在创建 URL 结构时保持一致性，而这种一致性完全没有必要，还可能给设计造成约束。资源以及资源间关系的标识可以完全独立于 URI 的结构，设计完成后再把 URI 空间映射到资源空间，这是使用超媒体驱动客户端的系统的独特优点。如果客户端基于服务器 URI 空间来构造 URI，那么设计者就会倾向于使用具有明显结构的统一的 URI 空间。

只要客户端能够理解我们在下一章中将讨论的媒体类型和链接关系，就可以使用这个问题跟踪服务，无需事先了解根资源以外的任何资源。

5.5　小结

本章关注的是问题跟踪应用程序的概念设计。我们回顾了构建可演化系统的初衷和需要付出的努力，定义了设计的组成部分，并对相关的应用领域进行了总结。

在根本上，我们要实现的是一个在不同系统间进行通信的分布式应用。为了使这个实现能够成功地进行演化，我们需要定义在系统组件之间传递应用语义的规范。这是下一章讨论的重点。

第 6 章

媒体类型选择与设计

订好协议，友谊常青。

我们常常听到开发人员说，基于 REST 的系统比别的 Web API 构建方法更好，是因为基于 REST 的系统更简单。人们经常认为，不使用协议是为了使系统简单，例如，WSDL（Web Service Description Language，Web 服务描述语言）就是如此。

但是，在构建分布式系统时，不可避免地要在组件之间预先进行一些约定。如果没有某种形式的共享知识，组件间就无法进行有意义的交互。

本章将介绍 Web 架构中使用的协议类型，如何选择符合我们需求的最佳协议，以及什么地方需要创建新的协议。

6.1 自描述

设计协议的一个关键概念是自描述。理想情况下，一个消息应该包含所有必要的信息，供消息接收者理解发送者的意图，或者，至少能提供所需信息的参考位置。

假设你收到一封信，信上写着"43.03384, –71.07338"。这封信提供了为完成某项具体目标所需的全部数据，但是你缺少实际应用这些数据所需要的上下文，从而无法使用。如果我告诉你，这一组数字是经纬度，那么你应该能理解这些数字的含义。显然，我假设你要么已经理解坐标系的概念，要么能够自己搜索到如何使用坐标系的信息。自描述并不是说消息需要真的包含经纬坐标系的描述，而只是需要用某种方式引用这个概念。

知道信中的信息是坐标，任务只完成了一半。应该怎么使用这些坐标呢？你还需要更多的

相关信息。如果你看到这封信的回邮地址写着“内华达州，畅饮好去处，邮政信箱 2000”，那么可能猜想到，这组坐标也许是一个推荐酒吧的地址。

6.2 协议类型

在 Web 的世界里，媒体类型用来表达一个资源表示什么，链接关系说明你为什么应对这个资源感兴趣。媒体类型和链接关系，就是我们在构建可演化系统时使用的协议。Web 架构中的媒体类型和链接关系代表着共享知识。在设计系统时，我们必须精心设计媒体类型和链接关系，如果媒体类型和链接关系发生改变，就会破坏依赖它们的组件。

6.3 媒体类型

媒体类型是独立于平台的类型，设计用于分布式系统间的通信。媒体类型用于传递信息，一个正式的规范定义了这些信息应该如何表示。

遗憾的是，Web API 远远没有发挥媒体类型的潜力。绝大多数的 Web API 只支持 `application/xml` 和 `application/json`。这两个媒体类型传递语义的能力极为有限，导致人们经常使用离线（out-of-band）知识来解释信息的内容。离线知识与自描述信息正好相反。回到我们刚才那个信件的例子，如果在你收到信之前，“畅饮好去处”公司已经告诉你，信上的数字会是地理坐标，那么我们就认为这个信息是离线知识。解释数据所需的信息是通过消息自身之外的手段传递的。如果使用通用类型，如 `application/xml` 和 `application/json`，我们就需要用别的方法来传达消息的语义。如果处理得不好，系统的演化就会变得更加困难，因为我们需要以离线方式传递系统发生的变化。使用离线知识时，客户端推测服务器将发送某种内容，如果服务器返回的内容改变，客户端就会无法自动适应这种变化。结果就是，客户端的存在锁定了服务器的行为。使用离线知识可能成为阻止系统演化的主要原因之一。

6.3.1 原始格式

这一节将给出几个示例，演示如何通过不同的媒体类型传递同一组数据。

示例 6-1 中展示的媒体类型 `application/octet-stream`，大概是你能使用的最基本的媒体类型。这个媒体类型是简单的字节流。接收到这个媒体类型，用户代理通常只是让用户把字节保存在文件中，别的什么也不能做。这个媒体类型没有定义任何应用程序语义。

示例 6-1：字节流

```
GET /some-mystery-resource
200 OK
Content-Type: application/octet-stream
Content-Length: 20

00 3b 00 00 00 0d 00 01 00 11 00 1e 00 08 01 6d 00 03 FF FF
```

示例 6-2 中展示的媒体类型 `text/plain` 告诉我们，消息内容可以稳妥地直接展示给最终用户，使用户能够读到这些数据。这个示例中的正文没有给出任何提示数据用途的信息。但是服务器完全可以在正文中加入一段文字，说明信息的用途。

示例 6-2：人工可读

```
GET /some-mystery-resource
200 OK
Content-Type: text/plain
Content-Length: 29

59,0,13,1,17,30,8,365,3,65535
```

谁了解我的业务？

过去 50 年间，业务应用程序的语义在系统的各处来回使用。在大型机时代，服务器了解数据的一切信息，客户端只是显示字符和检测按键的终端。

在 20 世纪 80 年代和 90 年代早期，随着个人计算机和局域网的兴起，客户端成了重头戏。虽然共享数据还是存储在文件服务器上，但是服务器只负责处理文件、行、列和索引。所有的智能处理都在客户端进行。

到 20 世纪 90 年末期，基于个人计算机网络的应用规模发展到了极限，客户端 / 服务器数据库开始流行，显示出了极大的可扩展潜力。

客户端 / 服务器数据库模式在驱动富客户端应用上只取得了有限的成功，不是因为技术上的问题，而是因为很多的开发者没有接受足够的训练，试图把用于 ISAM（Indexed Sequential Access Method，索引顺序访问方法）数据库的技术也应用在客户端 / 服务器数据库上。再加上提供商为获得可扩展性，大力推行客户端 / 服务器数据库，以直接取代原有的数据库系统，结果更加限制了富客户端的发展。

Web 应用程序在新世纪开始兴起。Web 应用把应用程序的工作流和业务逻辑放在了靠近数据的服务器上，取得了成功。这种方式解决了基于个人计算机的网络的大量通信问题，也避免了客户端 / 服务器数据库模式难以处理的一些并发问题。

近年来，JavaScript 从辅助基于 HTML 的 Web 体验，发展为创建和控制 Web 体验。一种趋势正在出现，将应用程序的工作流和逻辑移回客户端，但只限于 Web 浏览器的运行时环境。

如果我们要把业务逻辑移回客户端，就必须理解这种做法在过去为什么失败，然后才能避免重复先行者们犯过的错误。

在这个关键的架构选择中，具有超媒体能力的媒体类型是非常关键的一部分，这些媒体类型既能够传递工作流，也能够表达应用程序语义，因而工作负载能够在客户端和服务器之间进行更加合理地分布。

示例 6-3 中的媒体类型 `text/csv` 为返回信息提供了一些结构。这个数据模型定义为一组以逗号分隔的数值，随后拆分为（通常是）结构类似的数据行。我们还是不知道这些数据是什么，但是至少能够将数据格式化后展现给用户（假设用户知道自己看到的是什么）。

示例 6-3：简单结构化的数据

```
GET /some-mystery-resource
200 OK
Content-Type: text/csv
Content-Length: 29

59,0
13,1
17,30
8,365
3,65535
```

6.3.2　流行格式

请看示例 6-4。

示例 6-4：标记

```
GET /some-mystery-resource
200 OK
Content-Type: application/xml
Content-Length: 29

<root>
       <element attribute1="59" attribute2="0"/>
       <element attribute1="13" attribute2="1"/>
       <element attribute1="17" attribute2="30"/>
       <element attribute1="8" attribute2="365"/>
       <element attribute1="3" attribute2="65535"/>
</root>
```

在这个示例中，内容返回格式是 XML，这并不比 `text/csv` 格式具有更多语义。我们还是只有五对数字，XML 格式提供了给这些数据命名的地方。可是，这些名字的含义没有定义在 `application/xml` 格式的规范里，因此，如果任何客户端试图给这些名字赋予意义，还是需要依靠离线知识，从而引入了隐藏的耦合。在本章的后面部分，我们将讨论其他的方法，用于将语义附加在通用媒体类型之上，而又不创造隐藏的耦合。

对于更为复杂的场景，`application/xml` 格式可以用于表达数据的层次结构，为文本块标记附加信息。但是，`applicaiton/xml` 格式提供语义的方式十分有限，这个问题依然没有解决。

`application/json`（参见示例 6-5）沟通语义的能力甚至比 `application/xml` 的还要弱。在 Web 浏览器环境里使用 JSON 的好处是，我们可以下载 JavaScript 代码，为文档实现语义，从而使客户端和服务器能够同时演化。但是这种方法具有一个缺点，只能使用支持 JavaScript 运行时的客户端。这种方法也会影响中间层组件与消息交互的能力，因而无法

充分利用 HTTP 分层架构的优点。

示例 6-5：对象序列化

```
GET /some-mystery-resource
200 OK
Content-Type: application/json
Content-Length: 29

{ "objects" : [
       {"property1"="59", "property2"="0"},
       {"property1"="13", "property2"="1"},
       {"property1"="17", "property2"="30"},
       {"property1"="8", "property2"="365"},
       {"property1"="3", "property2"="65535"}
   ]
}
```

如果说通用类型位于媒体类型范围的一端，那么接下来要讨论的例子就位于与通用类型相反的另一端。在这个例子里，我们专门为某个应用程序定义了一个新的媒体类型，准确具备服务器理解的语义。为了让人们理解这个类型，我们需要为这个媒体类型编写一个规范，发布在因特网上，最好还要在 IANA 注册，这样的话，如果开发者希望理解收到的表达形式的含义，就能够很容易找到这个媒体类型的信息。

6.3.3 新格式

现在请看示例 6-6。

示例 6-6：服务相关的格式

```
GET /some-mystery-resource
200 OK
Content-Type: application/vnd.acme.cache-stats+xml
Content-Length: ??

<cacheStats>
       <cacheMaxAge percent="59" daysLowerLimit="0" daysUpperLimit="0">
       <cacheMaxAge percent="13" daysLowerLimit="0" daysUpperLimit="1">
       <cacheMaxAge percent="17" daysLowerLimit="1" daysUpperLimit="30">
       <cacheMaxAge percent="8" daysLowerLimit="30" daysUpperLimit="365">
       <cacheMaxAge percent="3" daysLowerLimit="365" daysUpperLimit="65535">
</cacheStats>
```

这个媒体类型终于说明了，我们一直在处理的数据原来是一个图形的数据点系列，用于展示因特网上请求消息的缓存控制标头中有效期长度值的分布。这个媒体类型提供了客户端处理这个信息的图形所需的所有信息。但是，这个媒体类型的应用场合非常特殊。人们多久会需要编写应用程序，展示缓存统计量的图形表示呢？如果要为这个媒体类型编制一个规范，并提交到 IANA 进行注册，似乎有点小题大做了。在今天，绝大多数的 Web API 创建这种使用范围很窄的有效载荷，但并不会费神编写规范和申请注册。但是，我们还有别的方法可以提供用户所需的所有信息，并且能适用于更多的场景。

请看示例 6-7 中的场景。

示例 6-7：领域相关的格式

```
GET /some-mystery-resource
200 OK
Content-Type: application/data-series+xml
Content-Length: ??

<series        xAxisType="range"
                       yAxisType="percent"
                       title="% of requests with their max-age value in days">
       <dataPoint yValue="59" xLowerValue="0" xUpperValue="0">
       <dataPoint yValue="13" xLowerValue="0" xUpperValue="1">
       <dataPoint yValue="17" xLowerValue="1" xUpperValue="30">
       <dataPoint yValue="8" xLowerValue="30" xUpperValue="365">
       <dataPoint yValue="3" xLowerValue="365" xUpperValue="65535">
</series>
```

在这个示例中，我们创建了一个媒体类型，用于传输绘制一个图形需要的一组数据点。绘制的图形可能是折线图、饼图和直方图，或者直接就是一个数据表。客户端可以理解这些数据点在绘制图形方面的语义。至于图形展示的是什么，需要人类用户来理解，客户端并不知道。但是，这些附加的语义使客户端可以进行一些操作，如：叠加图形、切换轴以及放大图形的某些部分。

这个媒体类型的可重用性比 `application/vnd.acme.cachestats+xml` 要高得多。任何需要图形化展示数据的应用场景都可以使用这个媒体类型。为这个媒体类型编写完整的规范所付出的努力，很快就能得到回报。

我认为，这种与领域相关，但不与服务相关的媒体类型，是媒体类型应该传递的语义的最佳平衡。这种媒体类型中，有些例子已证明颇为成功：

- HTML 用于传递超链接文本文档；
- Atom（参见 http://datatracker.ietf.org/doc/rfc4287/）设计用于整合基于 Web 的博客；
- AcitvityStream（参见 http://activitystrea.ms/）用于表示事件流；
- Json-home（参见 http://tools.ietf.org/html/draft-nottingham-json-home-03）设计用于发现 API 中可用的资源；
- Json-problem（参见 http://tools.ietf.org/html/draft-nottingham-http-problem-03）设计用于提供 API 返回错误的细节。

以上列举的所有这些媒体类型都有一些共同点。这些媒体类型都包含用于解决一个特定问题的语义，但是并不与任何特定的应用程序相关。每个 API 都需要返回错误信息；大部分的应用程序都有要用到事件流的领域。这些媒体类型的定义完全与平台及语言无关，也就是说任何开发者在任何类型的应用程序中都可以使用这些类型。还有很多新的媒体类型等待我们去定义，在很多的应用程序中重用。

6.3.4 超媒体类型

超媒体类型是一类媒体类型，通常基于文本，并包含指向其他资源的链接。通过在资源表示中提供链接，用户代理可以根据自己对链接含义的理解，从一个表示跳转到另一个表示。

超媒体类型对于客户端和服务器的分离，起到了极大的作用。通过使用超媒体，客户端不再需要事先了解 Web 上公布了哪些资源，而可以在运行时刻发现资源。

虽然超媒体在 HTML 中为 Web 应用提供的好处显而易见，但它在 Web API 开发中起到的作用却微不足道。因为缺乏工具，认为链接会不必要地增加资源表示的大小，以及普遍缺乏对超媒体优点的认同，Web 应用程序的开发者们往往会避免使用超媒体。

在有些情况下，例如：性能要求很高时，使用超媒体并不合适。对于性能要求高的系统，HTTP 协议可能也不是最佳选择。而当可演化性是一个基于 HTTP 系统的关键目标时，我们绝不能忽略超媒体的作用。

6.3.5 媒体类型爆炸

到这里，我们已经了解了通用媒体类型如何需要离线知识提供语义，也看到了携带领域相关语义的较为特殊的媒体类型示例。在 Web 开发社区中，一些人不愿意鼓励大家创建新的媒体类型，害怕产生“媒体类型爆炸”——创建大量的新媒体类型会产生设计糟糕的规范、重复工作和服务相关的类型，而且偶然重用的可能性会大大降低。这种担心不是没有根据的，媒体类型有可能会和物种一样演化，强者生存，弱者消亡。

6.3.6 通用媒体类型和档案

有些人喜欢采取另外一种使用媒体类型的方式。其基本做法是，使用一个较为通用的媒体类型，然后采取辅助方法在它的表示之上叠加语义。

这种方法的一个例子是 RDF（Resource Description Framework，资源描述框架，参见 http://www.w3.org/RDF/）媒体类型。简单说来，RDF 允许你使用三元组做出声明，即：主体、客体和谓词，其中谓词描述了主体和客体之间的关系。RDF 表示的语义中，主要部分由标准的谓词提供，谓词的含义有文档说明。RDF 规范将信息片段关联在一起，但是规范自身并没有定义任何具体的领域语法。

示例 6-8 是来自维基百科的 RDF 条目（参见 http://en.wikipedia.org/wiki/Resource_Description_Framework），其中的 URIhttp://purl.org/dc/elements/1.1 指向都柏林核心元数据启动计划（Dublin Core Metadata Initiative，DCMI）提供的一份单词表。

示例 6-8：RDF 示例

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
```

```
    xmlns:dc="http://purl.org/dc/elements/1.1/">
        <rdf:Description rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">
                <dc:title>Tony Benn</dc:title>
                <dc:publisher>Wikipedia</dc:publisher>
        </rdf:Description>
</rdf:RDF>
```

使用叠加语义方法的另一个例子是使用 ALPS（Application-Level Profile Semantics，应用层档案语义，参见 http://alps.io/spec/）。ALPS 用于指定可以应用在一个基础媒体类型（如 XHTML）上的领域语义，如示例 6-9 所示。使用最近成为标准的链接关系 `profile`，可以把这种额外的语义规范附加到已有的媒体类型之上。

示例 6-9：XHTML 上的 ALPS

```
GET /some-mystery-resource
200 OK
Content-Type: application/xhtml
Content-Length: 29

<html>
       <head>
       <link rel="profile" href="http://example.org/profiles/stats" />
   </head>
       <title>% of requests with their cache-control: max-age value in days </title>
       <body>
               <table class="data-series">
                       <thead>
                               <td>from</td>
                               <td>to (days)</td>
                               <td>percent</td>
                       </thead>
                       <tr class="data-point">
                               <td class="xLowerValue"></td>
                               <td class="xUpperValue">0</td>
                               <td class="yValue">59</td>
                       </tr>
                       <tr class="data-point">

                            <td class="xLowerValue">0</td>
                               <td class="xUpperValue">1</td>
                               <td class="yValue">13</td>
                       </tr>
                       <tr class="data-point">
                               <td class="xLowerValue">1</td>
                               <td class="xUpperValue">30</td>
                               <td class="yValue">17</td>
                       </tr>
                       <tr class="data-point">
                               <td class="xLowerValue">30</td>
                               <td class="xUpperValue">365</td>
                               <td class="yValue">8</td>
                       </tr>
                       <tr class="data-point">
                               <td class="xLowerValue">365</td>
```

```
                               <td class="xUpperValue"></td>
                               <td class="yValue">3</td>
                       </tr>
               </table>
        </body>
</html>
```

```
GET http://example.org/profiles/stats
200 OK
Content-Type: application/alps+xml

<alps version="1.0">
    <doc format="text">
        Types to support the domain of statistical data
    </doc>

   <descriptor id="data-series" type="semantic">
        <descriptor id="data-point" type="semantic">
         <doc>A data point</doc>
                <descriptor id="xValue" type="semantic"">
                        <doc>X value on graph</doc>
                </descriptor>
                <descriptor id="xLowerValue" type="semantic">
                        <doc>Lower bound on X range of values</doc>
                </descriptor>
                <descriptor id="xUpperValue" type="semantic">
                        <doc>Upper bound on X range of values</doc>
                </descriptor>
                <descriptor id="yValue" type="semantic" >
                        <doc>Y value on graph</doc>
                </descriptor>
                <descriptor id="yLowerValue" type="semantic">
                        <doc>Lower bound on Y range of values</doc>
                </descriptor>
                <descriptor id="yUpperValue" type="semantic">
                        <doc>Upper bound on Y range of values</doc>
                </descriptor>
                </descriptor>
   </descriptor>


</alps>
```

示例 6-10 中展示的 HAL（Hypermedia Appliation Language，超媒体应用程序语言，参见 http://stateless.co/hal_specification.html），是一种使用链接关系来应用领域语义的通用媒体类型。

示例 6-10：application/hal+xml 和 application/hal+json 中的 HAL

```
<resource       xAxisType="range"
                        yAxisType="percent"
                        title="% of requests with their max-age value in days">
        <resource       rel="http://example.org/stats/data-point"
                                yValue="59"
                                xLowerValue="0"
```

```
                                xUpperValue="0">
        <resource       rel="http://example.org/stats/data-point"
                                yValue="13"
                                xLowerValue="0"
                                xUpperValue="1">
        <resource       rel="http://example.org/stats/data-point"
                                yValue="17"
                                xLowerValue="1"
                                xUpperValue="30">
        <resource       rel="http://example.org/stats/data-point"
                                yValue="8"
                                xLowerValue="30"
                                xUpperValue="365">
        <resource       rel="http://example.org/stats/data-point"
                                yValue="3"
                                xLowerValue="365"
                                xUpperValue="65535">
</resource>
```

```
{
        "xAxisType" : "range",
        "yAxisType" : "percent",
        "title" : "% of requests with their max-age value in days",
        "_embedded" : {
                "http://example.org/stats/data-point" :
                { "yValue" : "59", "xLowerValue" : "0", "xUpperValue" : "0"},
                "http://example.org/stats/data-point" :
                { "yValue" : "13", "xLowerValue" : "0", "xUpperValue" : "1"},
                "http://example.org/stats/data-point" :
                { "yValue" : "17", "xLowerValue" : "1", "xUpperValue" : "30"},
                "http://example.org/stats/data-point" :
                { "yValue" : "8", "xLowerValue" : "30", "xUpperValue" : "365"},
                "http://example.org/stats/data-point" :
                { "yValue" : "3", "xLowerValue" : "365", "xUpperValue" : "65535"}
        }
}
```

HAL 依靠链接关系提供所需的语义，对资源表示中非 HAL 的部分进行解释。这意味着，链接关系类型 `http://example.org/stats/data-point` 的文档需要对 `yValue`、`xLowerValue` 和 `xUpperValue` 进行定义。HAL 并不关心这些值是属性还是元素，HAL 文档的使用者负责发现这些信息存储的具体位置。

使用链接关系来传递语义的一个问题是，端点 URI 经常没有链接关系。你如果在浏览器地址栏中输入一个链接地址，不会看到链接关系。对此有几种规避方法：你可以限制根资源，仅使用嵌入资源和链接；或者你可以使用链接关系 `type`，把语义联系到根资源。

使用通用的媒体类型的好处是，生成、解析和展示这些格式的工具很可能已经存在，可以重用。而且你可以定义语义档案，映射到多个不同的基础媒体类型。如果某个媒体类型更加适合在某个平台使用，就会更有优势。而在定义领域相关的媒体类型时，如果需要同时支持 XML 和 JSON 的类型变种，你就必须编写两份媒体类型规范，因为格式和语义都是

由媒体类型定义的。

人们正在试图规范描述语义档案的流程（参见 http://alps.io/spec/index.html），有可能会制定多个可行的方案。

使用通用媒体类型与辅助语义档案相结合的方法，有一个缺点是中间层组件难以了解消息的语义。HTTP 架构中的任何一层都可以轻易处理 `Content-Type` 标头中指定的媒体类型。如果使用如链接关系和档案的技术进行语义附加，中间层就难以发现语义信息，无法基于这些信息进行处理。在 Web 架构中，完成事情的方法通常不止一个，各种方法都有各自的优缺点，因此，我们必须合理设计系统，以满足需求。

辅助语义档案的使用是一个值得关注的领域。未来可期待出现更成熟的方法，进行消息语义的定义。

从前面列举的示例中，你可以看到，在客户端和服务器间传递同样的数据有很多种方法可用。有些媒体类型携带较多的语义信息，有些则较少。使用多少应用相关的语义驱动客户端，取决于是否存在标准的媒体类型满足你的需求，以及你对耦合性的容忍度。

6.3.7　其他超媒体类型

过去几年出现了一些新的超媒体类型。接下来的小节介绍了其中几个类型。

1. Collection+JSON

Collection+JSON 媒体类型适用于列表的应用领域。这个类型既通用又特殊，非常有趣。Collecion+JSON 只支持列表，但是并不关心列表中的东西是什么。这个类型还提供有趣的语义提示，可以描述如何查询列表以及添加新条目。这个类型对列表中的条目提供的语义虽然极为简单，但是支持档案标记，可用于描述列表中的条目。

2. Siren

Siren（参见 https://github.com/kevinswiber/siren）是另一个很新的超媒体类型，类似 HAL，很适合表示嵌套数据结构和嵌入资源。Siren 与 HAL 的不同之处在于，Siren 把语义附加在数据元素上。Siren 借用了 HTML 的 `class` 标记，用于标识语义信息。Siren 还对用于浏览资源的链接和代表行为的链接做了区分。`action` 链接也有自己的链接提示风格，告诉客户端如何调用这个 `action`。

虽然有人认为，大家应该统一使用一种格式，但是我情愿让自然选择产生结果，而不是试图强制所有的场景都使用一个超媒体格式。HTTP 使 API 更容易支持一个资源的多个表示，客户端可以从中进行选择，因此，使用多种格式不会成为系统发展的严重障碍。

到目前为止，我们主要把媒体类型作为一种沟通语义的方法进行讨论，但是正如在讨论 HAL 时提到的，语义也可以通过链接关系来传递。下一节将进一步讨论链接关系。

6.4 链接关系类型

在前面介绍协议类型的部分，我提到链接关系类型说明了你为什么可能对某个资源感兴趣。

链接关系类型是在 HTML 中首先引入的。最常见的链接关系类型大概就是 `rel="stylesheet` 了，这个值把一个 HTML 页面和辅助展示这个页面的样式表联系在了一起：

```
<link href="..." media="all" rel="stylesheet" type="text/css" />
```

不久之前，人们对链接关系类型进行了完整的定义，制定了 RFC 5988（参见 http://tools.ietf.org/html/rfc5988）规范。

6.4.1 语义

媒体类型承载的语义可以非常通用，链接关系也是如此。有些标准化的链接关系类型非常通用，如 `next`、`previous`、`current`、`item`、`collection`、`first` 和 `last`。但是，这些通用类型确实提供了相关资源的重要上下文信息。另一方面，有些标准化的链接关系的用途非常明确，如 `help`、`monitor`、`payment`、`license`、`copyright` 和 `terms-of-service`。

查看链接关系标准注册表（参见 http://www.iana.org/assignments/link-relations/link-relations.xml）并不能帮助我们完全了解链接关系的功能。大部分的标准链接关系都没有定义任何的行为或者约束，只是简单指明了目标资源，或者描述了上下文资源与目标资源之间的关系。

> **政　　治**
>
> 如果你对 Web 和因特网规范略加了解，立刻就会发现其中涉及很多政治因素。我得警告你，我本人倾向于主张由 IETF（Internet Engineering Task Force，因特网工程任务组）管理因特网相关的规范。IETF 采用 IANA 进行注册管理，因此我以 IANA 媒体注册表和 IANA 链接关系注册表为标准。但是，有些组织并不满意 IETF/IANA 的注册流程，而选择使用另一种链接关系的注册表（参见 http://microformats.org/wiki/existing-rel-values）。

有一些链接关系除了能标识关系，还开始给出更多的用途。让我们看看 `noreferrer` 和 `prefetch`。`noreferrer` 告诉用户代理，如果使用这个链接，就不应该指定 `referer` 标头；`prefetch` 告诉用户代理，在用户请求使用这个链接之前，用户代理应该先获取目标资源，使缓存中已有资源表示。在这两个例子中，链接关系实际上告诉了用户代理，服务器希望用户代理如何与这个链接进行交互。链接关系能提供更多的信息。一个链接关系类型的规范可以表明某个链接只使用 `POST` 方法，或者说明当访问链接时，返回的资源表示总是 `application/json` 格式的。

有些人选择不在链接关系类型的规范中定义交互机制，而是在链接中嵌入元数据，告诉客

户端如何与链接进行交互。例如：HTML 的 `<FORM>` 标签有一个方法属性，告诉浏览器使用 `GET` 或者 `POST` 方法。另一种方法是使用链接提示，服务器可以嵌入元数据，给出如何使用链接的建议。不过，这并不排除使用链接的其他有效方法。

所有这些方法都可以用于向客户端说明链接机制。可重用度最高的方法是，使用比较通用的链接关系嵌入元数据，但是这种方法传输的数据量最大，而且可能会减少客户端需要了解的链接关系数量。如果使用明确的链接关系，提供文档中定义的所有交互细节，那么使用的带宽较少，但是对客户端应用程序的要求更高。

如果把使用明确的链接关系这一想法发挥到极致，一个链接关系 `address` 可以要求使用 `GET` 方法，然后返回格式为 `application/json` 的资源，包括属性 `street`、`city`、`state`、`country`、和 `zipcode`。链接关系定义的约束越多，可重用性就越低，管理注册表的主题专家基本不可能接受这样的链接关系。但是，有一种*扩展链接关系类型*（extended link relation type），可以用 URI 作为标识符，唯一标识这个关系。使用扩展链接关系类型时，你不需要在 IANA 注册链接关系，可以随意进行定义。

使用链接关系精确描述预期的响应有一个有趣的效果，可以不依赖离线知识，使用如 `application/xml` 和 `application/json` 等通用类型传递数据。

但是，我个人的经验是，如果由链接关系和媒体类型均匀分担语义信息，产生的协议类型可重用性会更高。

链接关系和媒体类型的合作方式类似自然语言中的形容词和名词。形容词*快乐的*可以与很多名词一起使用。把独立的形容词和名词结合在一起，可以避免产生大量的新名词，如：*快乐狗*、*快乐牛*、*快乐鱼*，等等。

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/journal/?fromDate=20130921&toDate=20130922"
    "items" : [
      {
        "href" : "http://example.org/transaction/794",
        "data" : [
          {"amount" : "14576", "currency" : "USD", "date" : "20130921"}
        ],
        "links" : [
          {"rel" : "origin", "href" : "http://examples.org/account/bank1000"},
          {"rel" : "destination",
           "href" : "http://examples.org/account/payables/HawaiiTravel"}
        ]
      },
    "items" : [
      {
        "href" : "http://example.org/transaction/794",
        "data" : [
```

```
        {"amount" : "150000", "currency" : "USD", "date" : "20130922"}
      ],
      "links" : [
        {"rel" : "origin",
         "href" : "http://examples.org/account/receivables/acme"},
        {"rel" : "destination",
         "href" : "http://examples.org/account/bank1000"}
      ]
    }
  }
}
```

举个例子，假设我们要注册两个链接关系类型 `origin` 和 `destination`。在很多情况下，我们都需要表示某物从哪里来和到哪里去——不管是文件副本、银行转账还是地图上的一条道路。同样的链接关系可以重用在很多不同场景中。有时候，不管链接具体指向什么，这些可重用关系的语义就足以在客户端上实现通用的功能。这种特点和面向对象开发中的多形性颇为类似。

通常，当人们想到超媒体档案中的链接时，他们想到的是对领域概念之间的关系进行定义。在链接数据中，链接的主要用途就是定义概念之间的关系。但是，我们也可以使用链接实现更多功能，而不仅仅是声明域中的关系。

6.4.2　替换嵌入资源

在 HTML 世界中，我们习惯创建链接，指向图像、脚本和样式表。但是，我们很少见到 API 中公开这种静态资源。通过巧妙利用客户端的私有缓存，API 可以有效地提供各种信息的静态资源。通常情况下，这些信息原本都嵌入在客户端应用程序中。

6.4.3　间接层

链接可以提供一层间接性。如果我们在 API 的入口点创建发现文档，客户端可以动态发现特定资源的位置，无需把 URI 硬编码到客户端应用程序中（参见示例 6-11）。

示例 6-11：GitHub 的一个发现资源

```
GET https://api.github.com/
{
        "current_user_url":"https://api.github.com/user",
        "authorizations_url":"https://api.github.com/authorizations",
        "emails_url":"https://api.github.com/user/emails",
        "emojis_url":"https://api.github.com/emojis",
        "events_url":"https://api.github.com/events",
        ...
        "user_search_url":"https://api.github.com/legacy/user/search/{keyword}"
}
```

有了这个间接层，服务器可以重新组织 URI 结构，客户端无需进行改动。假设 GitHub 要

允许通过地址搜索用户，就可以在发现资源中做如下修改：

```
"user_search_url":"https://api.github.com/legacy/user/search{?keyword,country}"
```

假设客户端的设计遵循了 URI 模板（RFC 6570）的令牌替换（token replacement）规则，我们就不需要为适应这个新 URI 格式进行客户端改动。

我们也可以使用间接层提供一种智能负载均衡。如果某些资源给服务器带来了不合比例的大量负载，我们可以修改 URI，指向有容量的替代服务器。当不同类型的请求产生不同类型的有效载荷时，这种负载均衡方法会发挥一定的作用。

间接层还可以用于找到地理位置不同的资源。使用基于 IP 地址的客户端地址，响应消息中的表示可以包含链接，指向地理位置靠近客户端的服务器。远距离上的网络延迟可能很长，因而这种方法可能很重要。从位于美国西岸的客户端访问东岸的服务器，大约需要 80 毫秒。在有很多的资源可以获取以展示用户界面时，终端用户很快就能察觉到其存在。

6.4.4 引用数据

数据输入的界面常常向用户提供选项列表。这些列表不需要在客户端应用中预先定义，实际上，客户端甚至不需要事先知道哪个列表与某个输入字段相关。客户端应用程序可以用选项列表的链接标记一个输入字段，以此确定可用的条目列表，而无需对输入域有所了解。

例如：

```
<InputForm>
  <Street>/<Street>
  <City></City>
  <Province domainUrl="http://api.example.org/lists/provinces&country=CAN"/>
  <Country>Canada</Country>
</InputForm>
```

HTML 表单把整个列表嵌入在输入元素中，以实现类似的功能，但是这种方法效率并不高。使用链接可以减少有效载荷的大小。有时候，某些信息不经常使用，可以移到另一个资源中。如果不需要附加信息，你可以不提取这些信息，这样通常能够抵消二次通讯的额外开销。

数据易变性的差别是拆分资源的第二个原因。如果设备传感器读数的资源表示中包含设备的所有配置细节，会造成浪费，因为传感器读数比设备配置信息的变化要频繁得多。我们可以在资源表示中添加一个链接，指向设备配置细节，用户只在需要的时候使用即可。

拆分资源的第三个原因是重用。我们前面给出的地址 / 省份列表的例子就属于这种情况。对任何加拿大地址，省份列表都是一样的。如果一个用户要输入多个地址，那么利用客户端缓存中的省份列表就可以提高效率。

6.4.5 工作流

在基于 REST 的系统中，最独特的特征之一可能就是应用程序工作流的定义和沟通方式。在基于 RPC 的系统中，客户端必须理解应用程序的交互协议。例如：客户端必须知道，在调用 send 之前要调用 open，结束时要调用 close。这些规则必须事先在客户端实现，任何动态的状态检测都必须是定制的。

嵌入在资源表示中的链接，可以根据状态告诉客户端哪些交互是可行的。客户端必须知道所有交互的类型，以便能够使用这些交互，但是不必再费功夫去了解什么时候可以进行某种类型的请求。

请看示例 6-12，是和前面所举例子同样使用超媒体的场景。

示例 6-12：使用超媒体定义工作流

```
GET /deviceApi
200 OK
Content-Type: application/hal+xml

<resource>
        <link rel="http://example.org/rels/open" href="/deviceApi/sessions"/>
</resource>

POST /deviceApi/sessions
Content-Length: 0

201 Created Session
Content-Type: application/hal+xml
Location: http://example.org/deviceApi/session/1435

<resource>
  <link rel="http://example.org/rels/send" href="/deviceApi/session/1435{?message}"/>
  <link rel="http://example.org/rels/close" href="/deviceApi/session/1435"/>
</resource>

DELETE /deviceApi/session/1435
200 OK
```

虽然客户端还是需要理解 open/send/close 链接关系，但是服务器可以引导客户端完成工作流程。客户端必须知道：激活 open 链接，需要发送一个没有正文的 POST 请求；激活 close 链接，需要使用 DELETE 方法。在这个示例中，响应消息使用的是 HAL 格式，但是服务器完全可以返回多种超媒体格式。链接关系不必限制返回的媒体类型。但是，客户端至少需要理解服务器返回的其中一种媒体类型。

如果客户端设计为可使用动态的链接，那么客户端就可以适应工作流中的变化。例如：客户端使用一种算法，首先寻找 send 链接，如果没有找到 send 链接则寻找 open 链接，使用 open 链接，然后再寻找 send。采用这种方法，如果 API 的新版本不再需要使用 open/

close，那么我们就可以修改原有的 /deviceApi 资源表示，直接包含 send 链接。客户端能够自动适应这个新协议，不做改动就能继续工作。

这个示例非常的简单。复杂应用程序的交互协议会更为复杂，有更多的机会利用这种动态的工作流功能。

6.4.6 语法

RFC 5988 也定义了在 HTTP 标头中嵌入链接的格式，因此，即使是使用二进制内容（如图像和视频），你也可以在返回的资源表示中包含超媒体。但是，RFC 5988 没有定义链接应该如何嵌入其他的媒体类型。链接如何进行序列化，必须由媒体类型规范自己定义。像 application/json 和 application/xml 这样的媒体类型没有定义链接应该如何表示，因此使用起来可能会有问题。人们使用过一些不同的方法，但是因为缺少正式的规范，很难写出可重用的代码进行链接解析。有些争论看似微不足道，但是也需要定义，我就曾看到人们为了 JSON 对象应该叫作 links 还是 _links 而辩论好几个小时。

下面是一些链接语法的示例：

```
application/hal+json

"_links": {
       "self": { "href": "/orders" },
       "next": { "href": "/orders?page=2" },
       "find": {
           "href": "/orders{?id}",
           "templated": true
       },
       "admin": [{
           "href": "/admins/2",
           "title": "Fred"
       }]
    },

application/collection+json
  "links" : [
          {"rel" : "blog", "href" : "http://examples.org/blogs/jdoe",
                "prompt" : "Blog"},
          {"rel" : "avatar", "href" : "http://examples.org/images/jdoe",
                "prompt" : "Avatar", "render" : "image"}
          ]

application/vnd.github.v3+json

"assignee": {
      "login": "octocat",
      "id": 1,
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "somehexcode",
      "url": "https://api.github.com/users/octocat"
```

```
    }

application/hal+xml
<link rel="admin" href="/admins/5" title="Kate" />

application/atom+xml
<link href="http://www.example.org/data/q1w2e3r4" rel="related" hreflang="en" />
<collection href="http://example.org/blog/main" />
<content src="http://www.example.org/blog-posts/123" />
<icon>http://www.example.org/images/icon</icon>

text/html
 <link  rel="stylesheet" type="text/css"
        href="http://cdn2.sstatic.net/stackoverflow/all.css?v=c9b143e6d693">

 <a href="/faq">faq</a>

<form id="search" action="/search" method="get" autocomplete="off">
        <div>
            <input      autocomplete="off" name="q" class="textbox"
                        placeholder="search" tabindex="1" type="text"
                        maxlength="240" size="28" value="">
        </div>
</form>
```

从这些示例可以看到，超媒体表示中的链接可以采用多种形式。我希望在未来几年能看到更多的格式合并，减少一些表示上的差异。值得注意的是，虽然这些示例中很多都没有 rel 属性，但却具有链接关系类型的概念。以 HTML 为例，一个 <a> 标签可以很简单地表示为如下形式：

```
<link rel="a" href="/faq"/>
```

同样，<FORM> 标签可以表示为：

```
<link rel="form" id="search" action="/search" method="get" autocomplete="off">
        <div>
            <input      autocomplete="off" name="q" class="textbox"
                        placeholder="search" tabindex="1" type="text"
                        maxlength="240" size="28" value="">
        </div>
</link>
```

这两种风格只是演示了定义链接关系语法的两种不同方式。有些人在使用 RFC 5988 定义链接关系时会遇到一些问题，我们也可以使用这两种方式对此加以规避。按照 RFC 5988 的规定，为了使用一个简单的标记，如 rel="destination"，你必须在 IANA 注册这个关系，也就是说要接受一项由领域专家进行的审查。IANA 注册表的目的是为了鼓励人们开发适用于不同媒体类型中的链接关系。正如之前提到的，你可以使用扩展媒体类型的概念，创建一个使用 URI 的链接关系。但是，URI 可能会很长，在资源表示中过于花哨。如果你想创建一个专门供某个媒体类型使用的链接关系，那么可以选择使用如下的序列化：

```
<family>
  <mother href="/people/bob"/>
  <father href="/people/mary"/>
</family>
```

使链接关系类型成为媒体类型语法整体的一部分，你就明确声明了，这个链接关系只在这个媒体类型中定义，避免了在扩展链接关系类型中进行 URI 命名。

链接关系还有一个有趣的属性。链接可以指定多个关系，例如：

```
<link rel="first previous" href="/foo" />
<link rel="nofollow noreferrer" href="/bar" />
```

请注意，这个功能只是序列化的优化，但是的确允许链接进行行为组合。RFC 5988 规定：

> 关系类型不应当根据另一个关系类型存在与否，或者该关系类型自身出现的次数，来推断任何附加的语义。

在语义上，前一个示例和下面的示例并没有区别：

```
<link rel="first" href="/foo" />
<link rel="previous" href="/foo" />

<link rel="nofollow" href="/bar" />
<link rel="noreferrer" href="/bar" />
```

RFC 5988 还定义了链接的一组其他元数据属性，可以为用户代理提供信息，说明链接应该如何处理。这些信息不是在书面文档中指定，而是嵌入在链接的表示中：

```
<link href="..." rel="related" title="More info...." hreflang="en"
      type="text/plain" >
```

这些属性只是给用户代理的提示信息，并不能保证服务器会提供与提示信息一致的资源表示。

6.4.7　完美结合

链接关系类型和媒体类型就是分布式应用世界的花生酱和果酱。链接关系类型把资源表示绑定在一起，创造出完整的应用程序，帮助用户实现目标。当你把语义均匀分布在这些协议类型上，使重用成为可能时，这些类型工作效果最佳。

6.5　设计新的媒体类型协议

在试图确定用于某个资源的最佳媒体类型时，你应该总是首先考虑标准媒体类型。创建媒体类型具有挑战性，虽然有时标准媒体类型可能不是正好符合你的需求，但是也许能够承载足够多的语义信息，使客户端实现用户的目标。

如果你确定现有的媒体类型或链接关系不能承载所需的语义，那么也许可以考虑创建一个新的媒体类型。在创建媒体类型时，你需要设计实现如下特征。

- 表达的语义可以由一个以上应用程序使用。
- 要求最简的语法，进行开放性的假设。也就是说，信息的缺失并不说明该信息的任何情况。
- 不可识别的信息不予理会，除非该信息与其他规则冲突。

6.5.1 选择格式

当开发者们考虑为媒体类型选择一个格式时，大多数时候会想到用 XML 或 JSON 作为基础格式。选择 XML 或 JSON 作为基础格式的好处是，有很多处理这些格式的工具可用，而且 XML 或 JSON 为定义附加语义提供了灵活的结构。XML 和 JSON 各有优缺点，很多时候，选择哪个格式完全是个人喜好。但是，你打算支持哪种客户端类型的确会对格式选择产生影响。如果你的媒体类型预期的主要使用者是 JavaScript 客户端，那么显然应该选择 JSON。而对于与大型企业应用结合的系统，XML 可能更加合适。不管是哪种情况，作为 Web 开发者，两种格式我们都需要掌握，并使用最适合需求的格式。

但是，我要提醒你，同时使用 XML 和 JSON 要格外谨慎。XML 和 JSON 在数据表示方法上差异很大，如果两个格式一起使用，你创建出的格式有可能是最小公分母，不能充分发挥任何一方的优点。如果你的确需要同时支持这两种格式，那么就需要认识到，管理两种不同的规范格式需要双倍的工作量，而且使用这个媒体类型的用户可能也不多。对于像 HAL 这样的通用类型，支持 XML 和 JSON 两种变种可能还有意义，但是你应该谨慎做出决定：支持两个功能略有不同的变种可能会产生很多困惑，与其这样，还不如去迎合那些不采用单一格式的主流用户。

有时，XML 和 JSON 都不是合适的选择。很多时候，人们使用 JSON 文档去更新单个属性的值。在有些情况下，纯文本格式的表示是最简单的选择。所有的语言都提供程序库，把简单文本转换成本地数据类型。如果你只是要传输一个简单数值，那么可以考虑使用 `text/plain` 或者其衍生类型。



`text/plain` 格式的使用是一个很好的例子，说明了为什么我们应该把元数据放在 HTTP 表示的正文之外。很多 API 把状态码和其他元数据放在相应消息的正文里。但是，如果这样做，你能使用的媒体类型就受到了限制，而且重复了 HTTP 标头的含义。如果你不得不支持无法访问 HTTP 标头的客户端，那么可以专门为这些客户端定义特殊的媒体类型，而尽量不要在 API 中约束其他功能更强的客户端。

不止是基于文本的类型可以进行创造性的使用。在一篇博文（http://roy.gbiv.com/untangled/2008/paper-tigers-and-hidden-dragons）中，Roy Fielding 演示了如何把一个单色图

像用做稀疏数列，在一个表示中标识出已修改的资源，从而避免对大量资源进行轮询。

6.5.2 支持超媒体

正如第 1 章中介绍的，对于不是基于文本的媒体类型，超媒体的最佳选择是使用 RFC 5988 中定义的链接标头。对于基于文本的格式，我们在介绍链接关系时讨论了各种现有的链接语法。但是，在媒体类型层次，我们需要解决一些其他的问题。媒体类型应该允许一个关系有两个链接吗？如果允许，那么用户代理该如何区分这两个链接呢？在 HAL 中，一个链接可以由相关的 `name` 属性进行标识。HTML 的标签可以有相关的 `id` 属性，用做标识。

超链接需要指向这个媒体类型的一个实例的某个资源吗？我们应该为标识片段定义语法吗？

解析相对 URI 的规则是什么？

6.5.3 可选、强制、省略和适用

我发现，在设计媒体类型时，尤其是用于表示可写资源的媒体类型时，我们需要传递特定信息是否存在的语义。最明显的情况是 `mandatory` 元素。在我们的 `Issue item` 媒体类型中，`title` 属性是唯一的强制属性。

对于非强制的属性，资源表示中一个属性缺失的原因有很多种。一个资源可能出于性能上的考虑，选择只包含属性的一个子集，因此省略了某些属性。属性缺失的另一个可能原因是属性不适用。

*适用性*是指一个属性的相关性取决于资源中另一个属性的值。例如：在员工记录中，可能有一个 `TerminatedDate` 字段。如果一个员工的状态是 `Current`，那么 `TerminatedDate` 字段就很可能不适用。数据库的表和类不能按单个实体动态改变结构，因此我们经常会用 `null` 值表示一个属性不具备有意义的值。糟糕的是，null 值也用于表示一个属性还没有赋值，这种情况和属性不适用不是一回事。

在媒体类型表示中，我们可以完全省略与不适用属性相关的任何语法，只包含属性的语法，如果属性值还未定义，就设置为 `null` 或 `empty`。

使用这种从资源表示中移除不适用属性的策略，可以简化客户端代码，减少服务器和客户端的耦合。应用中经常有业务逻辑把控制属性和受控属性关联在一起。如果客户端假设一个属性的存在表示这个属性适用，那么客户端就不需要了解相关业务逻辑，业务逻辑可以进行演化而不影响客户端。



明确定义的媒体类型可以区分强制、适用和省略属性，而对象序列化格式受到类可表达语义的限制，这说明明确定义的媒体类型具有更好的表达能力。

在定义只包含资源子集属性的表示时，你需要明确区分省略信息和不适用信息。属性组可以区分强制字段，有时也同样可以帮助区分省略信息和不适用信息。

6.5.4 嵌入元数据和外部元数据

你可以使用注解在资源表示中包含元数据，如 `mandatory` 标志、类型定义和范围条件。例如：

```
<foo>
    <fooDate required="true" type="Date" minValue="2001/01/01"
                maxValue="2020/12/31">2010/04/12</fooDate>
</foo>
```

使用这种方法，元数据会与实际数据一起解析，客户端很容易访问元数据。但是，随着元数据量的增长，资源表示的大小会显著增加，而且元数据的变更通常远不如数据变更频繁。此外，同一个资源类的多个资源通常会重用同样的元数据。

如果一个资源包含两组生命周期不同的数据，最好的选择通常是将这个资源分解为两个资源，把生命周期较短的资源链接到生命周期较长的资源，使缓存层可以减少网络上传输的数据量。

使用外部元数据的一个挑战在于，我们必须说明哪些元数据适用于哪些表示数据。有些媒体类型定义了选择语法，可以指向文档中的具体数据片段。例如，CSS 样式表使用 `selectors`（参见 http://www.w3.org/TR/CSS2/selector.html），基于 XML 的表示可以使用 XPath 查询，定位文档中的节点。

6.5.5 可扩展性

媒体类型位于客户端和服务器耦合的交互点。媒体类型规范的破坏性变更可能会破坏客户端功能。如果我们确保媒体类型的可扩展，那么就可以适应变化的需求，同时减少破坏性变更的发生。编写处理可扩展格式的客户端代码，不能像编写普通客户端代码那样，对格式进行各种假设，因而更加困难。但是，为编写更灵活的解析代码所做的努力很快就能得到回报。

要获得可扩展性，一种常用的策略是忽略未知内容。对于有文档模式（如 XSD 结构定义）使用经验的人，这种策略看似有点违反常理。但是，使用这种策略，已有的解析器可以继续处理包含额外信息的新版媒体类型。我们的 `Issue` 极简信息模型，可以用 XML 进行如下表示：

```
<Issue>
   <Title>This is a bug</Title>
   <Description>Here are the details of the bug.</Description>
</Issue>
```

假设我们编写了一个解析器，使用 XPath 查询 `Issue/Title` 和 `/Issue/Descripton` 寻找文档元素。如果媒体类型进行了如下改进：

```
<Issue>
   <Title>This is a bug</Title>
   <FoundBy href='http://issueapi.example.org/user/342'/>
   <Description>Here are the details of the bug.</Description>
</Issue>
```

那么，现有的解析器，虽然会漏掉一些信息，但还能够继续处理这个文档。新版本中增加的信息该如何处理，很大程度上取决于具体情况。在有些情况下，忽略增加的信息也无妨；另外一些情况下，应该提醒用户，说明有些信息未能处理。有些服务可能拒绝接受服务其无法理解的额外信息。所有这些处理方法都是可行的，但是没有必要限制媒体类型只支持一种情况。

XSD 结构定义经常带来的另一个约束是文档元素的顺序。除非属性的顺序具有某种语义影响，否则媒体类型规范没有必要强行安排元素的顺序。属性以某种特定顺序出现，可能会简化解析逻辑，提高性能，但是，一旦出现未知属性，这些好处就微不足道了。支持可扩展性既需要应用约束，也需要避免不必要的约束。

媒体类型规范应该尽量使自己仅受限于应用域的约束，而不要受到服务实现的约束。例如：如果一个服务后台使用数据库，那么服务通常会定义字段长度。字段长度是服务使用的数据库的约束。使用这个媒体类型的其他应用实现很可能有着不同的物理约束。因为当前实现的限制，而任意规定一个最小的约束，这种做法既不必要，也不明智。

JSON 数值

关于 JSON 标准，人们还在进行着一场有趣的辩论。大部分的 JSON 实现中，JSON 文档的数值范围和 JavaScript 定义的数值限制是一样的（64 位浮点值）。但是，JSON 的创建者 Douglas Crockford 认为，JSON 应该独立于 JavaScript（参见 http://www.ietf.org/mail-archive/web/json/current/msg00308.html），JSON 文档中的数值表示应该没有限制。这是一种前瞻性的观点，认识到 JSON 很可能会比现在的 JavaScript 实现存在时间更长。不可否认，这个决定使解析器实现者的日子不好过，但是我相信这些工作都是值得的。

作为一个基本准则，在媒体类型中定义约束时，我会问自己一个问题：如果没有这个约束，是不是还可以解析资源表示，并传递同样的含义。如果回答是肯定的，那么我不会定义这个约束。规范中的规则越少，我们就越有可能在媒体类型中进行扩展，而不影响已有的代码。

6.5.6 注册媒体类型

为了让这个媒体类型的“分布式类型系统”发挥作用，人们需要一个方法发现存在哪些可用的类型。 IANA 注册表就是查找媒体类型的中心所在。但是，我们必须承认 IANA 媒体类型注册表的现状颇为混乱。目前，IANA 注册表由几个 Web 页面组成，页面上是一堆链接。很多这些链接指向的基本是一封 20 年前的电子邮件（参见 http://www.iana.org/assignments/media-types/application/atomicmail）。但是，这些条目至今仍然存在，说明了因特网上部署类型的持久性。一旦一个新类型发布到因特网上，就没有可靠的方法能删除这个类型。这也是创建媒体类型新版本为什么会带来问题的另一个原因，没有一个简便的方法能告诉人们“不要再使用那个版本”。

IANA 的很多注册条目已经更新为较新的 XML/XSLT/XHTML 格式，更容易由网络爬虫发现和理解。但是，媒体类型注册表还没有进行这一“修缮”，使用起来仍然很不方便（参见 http://roy.gbiv.com/untangled/2009/wrangling-mimetypes）。

媒体类型的注册流程也还非常原始。你需要阅读六个不同的 RFC 规范，然后提交一个 HTML 表单（参见 http://www.iana.org/form/media-types）。在提交申请表之前，你最好在因特网上发布所提议的规范，然后向 IETF 类型邮件列表（参见 http://www.ietf.org/mail-archive/web/ietf-types/current/maillist.html）发送一个声明，说明你打算递交这个提案。这个邮件列表中的专家可能会给你提供反馈意见，帮助解决提案中发现的问题。请注意，这些专家是为你义务提供指导，不是告诉人们如何进行媒体类型设计的。在交流过程中，请不要把大家的直率和简洁误认为是态度不友好！

IANA 的确需要一些来自各方的压力，促进其改进媒体类型注册表和注册流程。媒体类型注册表是因特网架构的核心组成部分，但是因为其粗糙的外表，给很多访问者留下了陈旧废弃的印象。

6.6 设计新的链接关系

如果搜索了链接关系注册表（参见 http://www.iana.org/assignments/link-relations/link-relations.xhtml），但是没有找到一个链接关系，能够描述你要链接的资源类型，那么你可以选择创建自己的链接关系类型。有三种不同的方式可以完成这个任务：

- 定义一个新的标准链接关系类型规范，提交审批；
- 创建一个“扩展的”链接关系类型，供自己使用；
- 如果你正在创建媒体类型规范，把链接规范集成到媒体类型规范中。

6.6.1 标准链接关系

创建一个标准链接关系的好处是，你可以在 `rel` 值中使用一个短名字。描述和注册一个链

接关系类型不一定很复杂。你可以参考 `licencse` 链接关系规范（参见 http://tools.ietf.org/html/rfc4946）的例子。

有意思的是，RFC 4946 规范只讨论了在 Atom 文档语境中使用 `license` 关系。IANA 链接关系注册表中也有一个链接，指向 HTML 规范中关于在 HTML 文档中使用 `license` 关系的讨论。另一个类似的例子是 `monitor` 链接关系，其规范暗示 `monitor` 链接只用于 SIP（Session Initiation Protocol，会话初始协议）。这些规范暗示链接关系的用途与特定媒体类型绑定，这样做并不好。我们还要能够把许可信息指派给 HTML 和 Atom 源以外的媒体类型，而且 SIP 也不是监控资源状态的唯一方法。

当工作中有很多规定时，我们面临的一个挑战就是，知道何时必须遵守规定，何时可以打破规定。在前面提到的场景中，我相信这些链接关系的价值并不限于它们定义的语境，并打算在其他场景中使用这些链接关系。我希望，随着更多的人在新的场景中使用这些链接关系，人们能够更好地认识到链接关系的可重用性，在新的规范中避免将链接关系局限于某些具体媒体类型。

创建新链接关系的指导原则和创建新媒体类型的原则非常类似。链接关系应当尽可能的通用，同时又提供足够的语义，解决一个特定领域的问题。有些链接关系现在尚未成为标准，但是很有可能成为标准。

- Owner

  一个链接，指向主管当前资源的资源。

- Home

  一个指向 API 入口或者根资源的链接。

- Like

  一个不安全链接，说明用户对一个资源的喜爱度。

- Favorite

  一个不安全链接，请求将资源存储为用户的收藏。

Microformats 网站（参见 http://microformats.org/wiki/existing-rel-values）记录了很多其他的链接关系提案，以及一些得到使用但是未成为标准的链接关系。`sitemap`（http://microformats.org/wiki/rel-sitemap）是个广泛使用而有趣的链接关系，描述了预期响应的确切格式。`sitemap` 是把所有语义放置在链接关系中，但在媒体类型中不包含语义的例子。

6.6.2 扩展链接关系

RFC 5988 定义了扩展链接关系，用于创建未在 IANA 注册的链接关系。为了避免命名冲突，你必须使用 URI 做关系名。URI 利用域命名系统确保唯一性。不幸的是，在链接关系

中使用 URI，会使资源表示变得大而难看。你可以使用 CURIE（参见 http://www.w3.org/TR/2007/WD-curie-20070307/）缩写链接关系，但是，CURIE 看上去象 XML 命名空间而工作方式不同，因此有的人不喜欢使用 CURIE。

扩展链接关系非常有用，但是也可能导致滥用链接关系。一旦开发者意识到链接关系的威力，往往会使用过度，开始创建服务相关的链接关系。虽然这样做短期内没有问题，但是服务相关的链接关系引入了服务相关的耦合，并非系统演化和整体 Web 生态系统的最佳选择。

6.6.3 嵌入链接关系

如果链接关系与媒体类型的语义紧密相关，那么我们也许可以使链接关系成为媒体类型规范的一部分，只在这个媒体类型中使用。HTML 的 `<FORM>` 标签就是媒体类型自身中定义的链接关系的例子。但是，`<FORM>` 标签定义在 HTML 之中并不合理，现在人们需要在所有其他的超媒体类型中复制类似的功能。如果 `<FORM>` 定义为一个独立的链接关系，那么其他媒体类型要重用 `<FORM>` 会更加容易。

6.6.4 注册链接关系

注册链接关系的流程相当简单，在 RFC 5988（参见 http://tools.ietf.org/html/rfc5988）中有详细说明。

6.7 问题跟踪域中的媒体类型

在第 4 章中，我们确定了几个不同的资源分类：list 资源、item 资源、discovery 资源和 search 资源。对于每个资源分类，我们需要确定哪种媒体类型最适合承载所需的语义。

同质 API

一些开发者往往希望选择一种媒体类型，在整个 API 中重复使用。人们认为，只使用一种媒体类型会减少客户端开发者的工作。在很多时候，事实和人们认为的正好相反。试图把一个相当规模的 API 的所有语义都打包在一个媒体类型中，意味着要么媒体类型规范非常复杂，要么有些语义需要进行离线沟通。当 API 开始把链接与外部系统集成时，限制客户端只处理一种媒体类型就会带来问题。如果客户端设计为只处理指定的 API 媒体类型，那么可能很难支持其他 API 使用的其他媒体类型。

如果客户端能够轻易处理很多不同的媒体类型，就可以鼓励偶然重用，辅助系统演化和集成。

6.7.1　list资源

对返回条目列表的资源，我们将使用 `collection+json`（参见 http://amundsen.com/media-types/collection/）媒体类型。`collection+json` 是一个支持超媒体的类型，专门设计用于支持条目列表。这个媒体类型支持将任意一组数据与列表中的每个条目关联。`collection+json` 支持查询，可以搜索条目的各种子集，还包含一个模板属性，用于创建立列表中的新条目。

我们原本可以使用 HAL 甚至 XHTML，这两种类型都可以表示条目列表。但是，`collection+json` 是专为表示列表而设计的，似乎更加合适。示例 6-13 展示了如何使用 `collection+json` 表示一个问题列表。

示例 6-13：问题列表示例

```
{
  "collection": {
    "href": "http://localhost:8080/Issue",
    "links": [],
    "items": [
      {
        "href": "http://localhost:8080/issue/1",
        "data": [
          {
            "name": "Description",
            "value": "This is an issue"
          },
          {
            "name": "Status",
            "value": "Open"
          },
          {
            "name": "Title",
            "value": "An issue"
          }
        ],
        "links": [
          {
            "rel": "http://webapibook.net/rels#issue-processor",
            "href": "http://localhost:8080/issueprocessor/1?action=transition"
          },
          {
            "rel": "http://webapibook.net/rels#issue-processor",
           "href": "http://localhost:8080/issueprocessor/1?action=close"
          }
        ]
      },
      {
        "href": "http://localhost:8080/issue/2",
        "data": [
          {
            "name": "Description",
            "value": "This is a another issue"
          },
```

```
          {
            "name": "Status",
            "value": "Closed"
          },
          {
            "name": "Title",
            "value": "Another Issue"
          }
        ],
        "links": [
          {
            "rel": "http://webapibook.net/rels#issue-processor",
            "href": "http://localhost:8080/issueprocessor/2?action=transition"
          },
          {
            "rel": "http://webapibook.net/rels#issue-processor",
            "href": "http://localhost:8080/issueprocessor/2?action=open"
          }
        ]
      }
    ],
    "queries": [
      {
        "rel": "http://webapibook.net/rels#search",
        "href": "/issue",
        "prompt": "Issue search",
        "data": [
          {
            "name": "SearchText",
            "prompt": "Text to match against Title and Description"
          }
        ]
      }
    ],
    "template": {
     "data": []
    }
  }
}
```

6.7.2 item资源

对于单个问题的表示，我们有好几种选择。我们可以使用 HAL，定义一个链接关系 `issue` 说明内容；也可以使用 XHTML，定义一个语义档案，对 HTML 添加问题跟踪域的语义标记；或者可以定义一个新的媒体类型，用于问题表示。

问题的概念非常通用，很容易在其他服务中重用，因此为其创建一个新的媒体类型也是合理的。问题跟踪不是小众冷门的领域，每个软件开发者都会用到问题跟踪，很多客户支持呼叫中心也会使用。即便不同的实施导致不能进行全真通信，一个可以互操作的格式也可能会非常地有用。

示例 6-14 展示了这种媒体类型的一个表示样例。目前，这个媒体类型定义为 JSON。Web 技术的早期接纳者可能更熟悉 JSON 的使用，如果这个媒体类型受到关注，那么我们可以定义一个 XML 变种，扩大使用者范围。

附录 E 提供了这个媒体类型的完整规范。

> **支持多种格式**
>
> API 问题文档（参见 http://tools.ietf.org/html/draft-nottingham-http-problem-04）展示了一个有意思的方法，用于避免为 XML 和 JSON 格式变种创建两个不同的规范。在示例中，核心规范假定为 JSON 格式，但是规范的附录定义了如何将媒体类型映射到 XML 格式。

示例 6-14：问题示例

```
{
  "id": "1",
  "title": "An issue",
  "description": "This is an issue",
  "status": "Open",
  "Links": [
    {
      "rel": "self",
      "href": "http://localhost:8080/issue/1"
    },
    {
      "rel": "http://webapibook.net/rels#issue-processor",
      "href": "http://localhost:8080/issueprocessor/1?action=transition",
      "action": "transition"
    },
    {
      "rel": "http://webapibook.net/rels#issue-processor",
      "href": "http://localhost:8080/issueprocessor/1?action=close",
      "action": "close"
    }
  ]
}
```

6.7.3 discovery资源

discovery 资源是入口资源，指向系统提供的其他资源。对于这种资源，我们将使用一个新近提出的媒体类型：`json-home`（参见 http://tools.ietf.org/html/draft-nottingham-json-home-03）。这个媒体类型是专门设计用于为资源动态发现的入口资源提供表示。`json-home` 类似 Atom 服务文档（参见 http://tools.ietf.org/html/rfc5023#section-8），但是并不限于指向 Atom 源。`json-home` 文档可以包含指向任意资源的链接，并包含用附加的元数据，用于发现如何使用这些链接。示例 6-15 展示了问题跟踪 API 可能使用的一个 `json-home` 文档。

示例 6-15：根资源示例

```
{
  "resources": {
    "http://webapibook.net/rels#issue": {
      "href": "/issue/{id}",
      "hints": {
        "allow": [
          "GET"
        ],
        "formats": {
          "application/json": {},
          "application/vnd.issue+json": {}
        }
      }
    },
    "http://webapibook.net/rels#issues": {
      "href": "/issue",
      "hints": {
        "allow": [
          "GET"
        ],
        "formats": {
          "application/json": {},
          "application/vnd.collection+json": {}
        }
      }
    },
    "http://webapibook.net/rels#issue-processor": {
      "href": "/issueprocessor/{id}{?action}",
      "hints": {
        "allow": [
          "POST"
        ]
      }
    }
  }
}
```

6.7.4 search资源

对于搜索，我们希望能够使用 `collection+json` 的查询功能。如果 `collection+json` 的查询不能满足需求，我们会尝试使用链接关系的 `search` 和 OpenSearch（http://www.opensearch.org/）定义的协议。

6.8 小结

媒体类型和链接关系是用于管理分布式应用组件之间耦合的工具。本章介绍了使用这种耦合进行应用程序语义沟通的不同方法。如果你了解了现有规范，并知道如何以及何时创建新媒体类型和链接关系，就准备好开始实际构建 API 了。在下一章中，我们将使用已经掌握的知识，开始编写一个 API 示例。

第 7 章 构建API

空谈不如实践。

在前两章，你了解了问题跟踪系统的设计，以及用于系统交互的媒体类型。这一章，你将了解如何构建支持这一设计的 Web API 的基本实现。我们的目的不是要创建一个功能完整的 API，也不是要实现整个设计，而是实现系统的主要部分，在此之上才能关注其他，促进系统的演化。

这一章不会过于详细地介绍系统的任何一个独立部分，而是关注如何将各部分结合在一起。随后的章节将更为详尽地介绍 ASP.NET Web API 的每个不同方面。

7.1 设计

系统的设计大体如下。

(1) 有一个管理问题的后台系统（如 GitHub）。

(2) `Issue collection` 资源从后台获取资源，返回 `Issue+Json` 或 `Collection+Json` 格式的响应。`Issue collection` 资源也可以通过 HTTP 的 `POST` 方法创建新问题。

(3) `Issue item` 资源包含后台系统中一个问题的表示形式。问题可以通过 `PATCH` 方法更新，通过 `DELETE` 请求删除。

(4) 每个问题包含具有如下 `rel` 值的链接。

- `self`：包含资源自身的 URI。
- `open`：请求将问题状态改为 `Open`。

- `close`：请求将问题状态改为 `Closed`。
- `transition`：请求将问题转移到下一个合适的状态（例如：从 `Open` 转到 `Closed`）。

(5) 一组 `Issue Processor` 资源负责处理与问题状态转移相关的操作。

7.2 获得源代码

你可以从 WebApiBook 库下载问题跟踪系统 Web API 的实现（http://bit.ly/web-api-implement[1]）和单元测试代码，或者复制 issuetracker 库（https://github.com/webapibook/issuetracker），签出 BuildingTheApi 分支。

7.3 使用行为驱动开发构建实现

我们使用 BDD（Behavior-Driven Development，行为驱动开发，参见 http://dannorth.net/introducing-bdd/）风格的验收测试，采用测试驱动方法进行 API 的构建。这种做法和传统测试驱动开发风格的主要不同之处在于，其焦点是端到端场景，而不是具体实现。使用验收风格的测试，你可以了解从初始请求开始的整个端到端流程。

BDD 入门

行为驱动开发是 TDD（Test-Driven Development，测试驱动开发）的一种风格，关注系统行为的验证；而传统测试驱动开发关注的是不同组件的实现。在行为驱动开发中，需求通常由业务专家以特定格式编写，然后可由开发者执行。

行为驱动开发有各种不同的形式，但是大多数都使用 Gherkin 语法（参见 http://behat.readthedocs.org/en/v2.5/guides/1.gherkin.html），或 Given-When-Then 语法。这种语法把测试分解为功能和场景。一个功能是一个待测试的组件。每个功能有一个或多个场景，覆盖功能的不同部分。每个场景再按步骤细分，每个步骤都是一个 Given、When 和 Then，以及 And 或 But 声明。

Given 子句设置系统的初始状态，When 子句说明对系统的操作，Then 子句对预期的行为进行断言。每个子句都可以有多个部分，由 And（表示包括）或 But（表示排除）连接在一起。

7.4 浏览解决方案

打开 src 文件夹下的 WebApiBook.IssueTrackerApi.sln 文件，你将看到如下项目。

注 1：亦可登录 iTuring.cn 至本书页面下载。——编者注

- `WebApiBook.IssueTrackerApi`

  包含 API 的实现代码。

- `WebApiBook.IssueTrackerApi.AcceptanceTests`

  包含行为驱动开发的验收测试，用于验证系统的行为。在这个项目中，你将看到一个 Features 文件夹，其中包含对应每个功能的测试文件，每个文件包含一个或多个测试。

- `WebApiBook.IssueTrackerApi.SelfHost`

  包含 API 的自托管代码。

7.5 软件包和程序库

在代码中，你会看到如下软件包和工具：

- `Microsoft.AspNet.WebApi.Core`

  ASP.NET Web API 用于编写和托管我们的 API。Core 软件包提供了所需的最小功能集。

- `Microsoft.AspNet.WebApi.SelfHost`

  这个软件包提供了在 IIS 之外托管 API 的功能。

- `Autofac.WebApi`

  Autofac 用于依赖和生命周期管理。

- `xunit`

  XUnit 用作测试框架 / 运行器。

- `Moq`

  Moq 用于在测试中模拟对象。

- `Should`

  Should 程序库用于进行 Should 断言。

- `XBehave`

  XBehave 程序库用于测试中的 Gherkin 风格语法。

- `CollectionJson`

  支持 `Collection+Json` 媒体类型。

7.6 自托管

源代码中包含了 Issue Tracker API 的一个自托管程序。这个自托管程序可以启动 API，让

你使用浏览器或工具（如 Fiddler）向 API 发送 HTTP 请求。自托管使 ASP.NET Web API 易于开发，是一个很好的功能。打开应用程序（要使用管理员权限）运行，你马上就会看到，一个宿主已经启动并且正在运行，如图 7-1 所示。

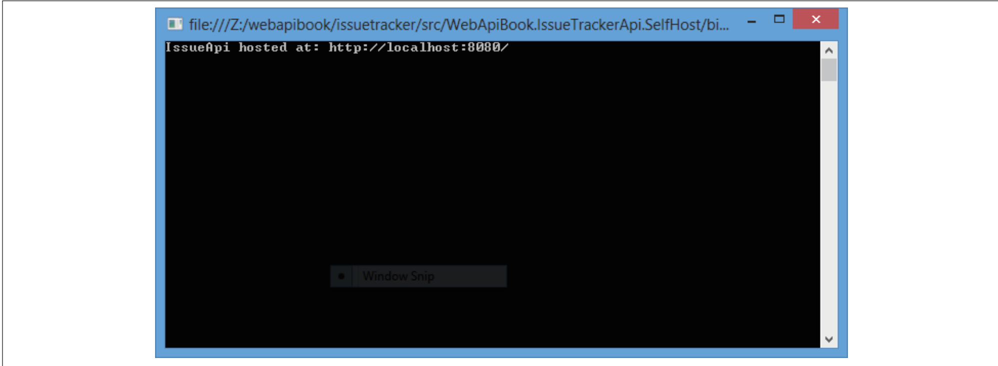


图 7-1：自托管

需要注意的是，如果在 Visual Studio 中运行自托管项目，你需要以管理员身份运行，或者使用 `netsh` 命令预留一个端口。

使用 `Accept` 标头值 `application/vnd.image+json`，向 http://localhost:8080 发送一个请求，就可以得到如图 7-2 所示的问题集合。

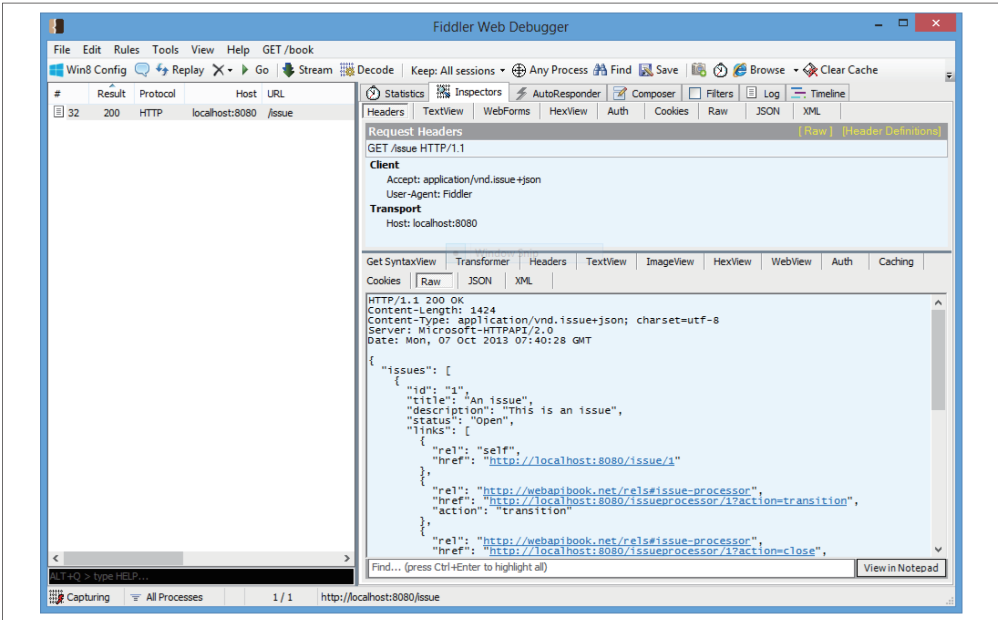


图 7-2：向自托管 API 发送一个获取问题的请求

如果在阅读这一章时，你想直接运行这个 API，就请使用这个自托管程序！你可以在 API 中设置断点，单步执行，了解实际执行了什么操作。

现在，开始动手实现 API！

7.7　模型和服务

Issue Tracker API 的实现依赖于一组核心服务和模型。

7.7.1　问题和问题库

因为这是一个问题跟踪项目，我们需要有地方存储和获取问题。接口 `IIssueStore`（WebApiBook.IssueTrackerApi\Infrastructure\IIssueStore.cs）定义了用于问题创建、获取和保存的方法（参加示例 7-1）。请注意，所有的方法都是异步的，因为这些方法可能受到网络 I/O 的限制，不应该阻塞应用程序的线程。

示例 7-1：`IIssueStore` 接口

```
public interface IIssueStore
{
    Task<IEnumerable<Issue>> FindAsync();
    Task<Issue> FindAsync(string issueId);
    Task<IEnumerable<Issue>> FindAsyncQuery(string searchText);
    Task UpdateAsync(Issue issue);
    Task DeleteAsync(string issueId);
    Task CreateAsync(Issue issue);
}
```

示例 7-2 中的 `Issue` 类是一个*数据模型*，包含了存储库中问题的持久数据。`Issue` 类只包含*资源状态*，不包含任何链接。链接是 API 层级的关注点，因此是*应用程序状态*，不属于应用域。

示例 7-2：`Issue` 类

```
public class Issue
{
    public string Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public IssueStatus Status { get; set; }
}

public enum IssueStatus {Open, Closed}
```

7.7.2　IssueState

示例 7-3 中的 `IssueState` 类（WebApiBook.IssueTrackerApi\Models\IssueState.cs）是一个状态模型，设计用于表示资源和应用程序状态。`IssueState` 可以在 HTTP 响应中以一种或多

种媒体类型表示。

示例 7-3：IssueState 类

```
public class IssueState
{
    public IssueState()
    {
        Links = new List<Link>();
    }

    public string Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public IssueStatus Status { get; set; }
    public IList<Link> Links { get; private set; }
}
```

请注意，IssueState 类和 Issue 类成员相同，但多了一个链接集合。你可以能奇怪为什么 IssueState 不继承 Issue 类，答案是为了更好地隔离关注点。如果 IssueState 继承 Issue 类，那这两个类就紧密耦合，Issue 的任何变化都会影响 IssueState。将关注点隔离，系统的一部分可以独立于另外的部分进行修改，符合我们对系统的可演化性期望。

7.7.3 IssuesState

示例 7-4 中的 IssuesState 类（WebApiBook.IssueTrackerApi\Models\IssuesState.cs）用于返回一个问题集合，这个集合包含一组顶级链接。请注意，这个集合还明确实现了 CollectionJson 程序库的 IReadDocument 接口。正如你将看到的，如果客户端发送的请求的 Accpept 标头值为 application/vnd.collection+json，CollectionJsonFormatter 就会使用 IReadDocument 接口输出 Collection+Json 格式的文档。而标准格式化程序会使用公共方法。

示例 7-4：IssuesState 类

```
using CJLink = WebApiContrib.CollectionJson.Link;

public class IssuesState : IReadDocument
{
    public IssuesState()
    {
        Links = new List<Link>();
    }

    public IEnumerable<IssueState> Issues { get; set; }
    public IList<Link> Links { get; private set; }

    Collection IReadDocument.Collection
    {
        get
        {
            var collection = new Collection(); // <1>
            collection.Href = Links.SingleOrDefault(l => l.Rel ==
```

```
                IssueLinkFactory.Rels.Self).Href; // <2>
            collection.Links.Add(new CJLink {Rel="profile",
                Href = new Uri("http://webapibook.net/profile")}); // <3>
            foreach (var issue in Issues) // <4>
            {
                var item = new Item(); // <5>
                item.Data.Add(new Data {Name="Description",
                    Value=issue.Description}); // <6>
                item.Data.Add(new Data {Name = "Status",
                    Value = issue.Status});
                item.Data.Add(new Data {Name="Title",
                    Value = issue.Title});
                foreach (var link in issue.Links) // <7>
                {
                    if (link.Rel == IssueLinkFactory.Rels.Self)
                        item.Href = link.Href;
                    else
                    {
                        item.Links.Add(new CJLink{Href = link.Href,
                            Rel = link.Rel});
                    }
                }
                collection.Items.Add(item);

            }
            var query = new Query {
                Rel=IssueLinkFactory.Rels.SearchQuery,
                Href = new Uri("/issue", UriKind.Relative),
                     Prompt="Issue search" }; // <8>

            query.Data.Add(
                new Data() { Name = "SearchText",
                    Prompt = "Text to match against Title and Description" });
            collection.Queries.Add(query);
            return collection; // <9>
        }
    }
}
```

这段代码中最有意思的逻辑是 `Collection`，这段逻辑生成一个 `Collection+Json` 文档：

- 实例化一个新的 `Collection+Json Collection` 集合。<1>
- 设置集合的 `href` 值。<2>
- 给集合添加一个指向集合描述的档案链接。<3>
- 重申 `IssuesState` 集合 <4>，创建对应的 `Collection+Json Item` 实例 <5>，并设置其 `Data`<6> 和 `Links` 值 <7>。
- 创建一个“Issue search”查询，添加到文档的查询集合中 <8>。
- 返回集合 <9>。

7.7.4　Link

示例 7-5 中的 Link 类（WebApiBook.IssueTrackerApi\Models\Link.cs）保存之前介绍过的标准 Rel 和 Href，并包含附加的元数据，描述与链接相关的可选操作。

示例 7-5：Link 类

```
public class Link
{
    public string Rel { get; set; }
    public Uri Href { get; set; }
    public string Action { get; set; }
}
```

7.7.5　LinkeStateFactory

现在系统有了 Issue 和 IssueState，还需要有一个从 Issue 获得 State 的方法。示例 7-6 中　的 IssueStateFactory（WebApiBook.IssueTrackerApi\Infrastructure\IssueStateFactory.cs）以 Issue 实例为输入参数，返回一个对应的包含链接的 IssueState 实例。

示例 7-6：IssueStateFactory 类

```
public class IssueStateFactory : IStateFactory<Issue, IssueState> // <1>
{
    private readonly IssueLinkFactory _links;

    public IssueStateFactory(IssueLinkFactory links)
    {
        _links = links;
    }

    public IssueState Create(Issue issue)
    {
        var model = new IssueState // <2>
            {
                Id = issue.Id,
                Title = issue.Title,
                Description = issue.Description,
                Status = Enum.GetName(typeof(IssueStatus),
                    issue.Status)
            };

        // 添加超媒体
        model.Links.Add(_links.Self(issue.Id)); // <2>
        model.Links.Add(_links.Transition(issue.Id));

        switch (issue.Status) { // <3>
            case IssueStatus.Closed:
                model.Links.Add(_links.Open(issue.Id));
                break;
            case IssueStatus.Open:
                model.Links.Add(_links.Close(issue.Id));
                break;
```

```
            }

            return model;
        }
    }
```

这段代码的工作机制如下：

- 这个工厂类实现 `IStateFactory<Issue, IssueState>` 接口，调用者可以依赖接口而非具体实现，因此更容易在单元测试中模拟这个工厂类；
- `Create` 方法初始化一个 `IssueState` 实例，复制 `Issue` 中的数据 <1>；
- 之后的代码包含业务逻辑，在 `IssueState` 实例中添加标准链接，如 Self 和 Transition <2>，以及上下文相关的链接，如 Open 和 Close <3>。

7.7.6 LinkFactory

虽然 `StateFactory` 包含添加链接的逻辑，但 `IssueLinkFactory` 负责创建链接对象自身。`IssueLinkFactory` 为每个链接提供强类型的访问方法，以提高调用代码的可读性和可维护性。

我们首先要介绍的是示例 7-7 中的 `LinkFactory` 类（WebApiBook.IssueTrackerApi\Infrastructure\LinkFactory.cs），其他的工厂类都派生自 `LinkFactory` 类。

示例 7-7：`LinkFactory` 类

```
public abstract class LinkFactory
{
    private readonly UrlHelper _urlHelper;
    private readonly string _controllerName;
    private const string DefaultApi = "DefaultApi";

    protected LinkFactory(HttpRequestMessage request, Type controllerType) // <1>
    {
        _urlHelper = new UrlHelper(request); // <2>
        _controllerName = GetControllerName(controllerType);
    }

    protected Link GetLink<TController>(string rel, object id, string action,
        string route = DefaultApi) // <3>
    {
        var uri = GetUri(new { controller=GetControllerName(
            typeof(TController)), id, action}, route);
        return new Link {Action = action, Href = uri, Rel = rel};
    }

    private string GetControllerName(Type controllerType) // <4>
    {
        var name = controllerType.Name;
        return name.Substring(0, name.Length - "controller".Length).ToLower();
    }
```

```
    protected Uri GetUri(object routeValues, string route = DefaultApi) // <5>
    {
        return new Uri(_urlHelper.Link(route, routeValues));
    }

    public Link Self(string id, string route = DefaultApi) // <6>
    {
        return new Link { Rel = Rels.Self, Href = GetUri(
            new { controller = _controllerName, id = id }, route) };
    }

    public class Rels
    {
        public const string Self = "self";
    }
}

public abstract class LinkFactory<TController> : LinkFactory // <7>
{
    public LinkFactory(HttpRequestMessage request) :
        base(request, typeof(TController)) { }
}
```

这个工厂类根据路由值和一个默认路由名，生成 URI。

- 这个工厂类的构造函数有两个参数：一个参数 `HttpRequestMessage` <1> 用于构造一个 `UrlHelper` 实例 <2>；另一个参数控制器类型用于生成一个“self”链接。
- `GetLink` 泛型方法使用一个 rel，一个控制器以及其他参数生成一个链接。<3>
- `GetControllerName` 方法按照给定的类型获取控制器名，由 `GetLink` 方法使用。<4>
- `GetUri` 方法使用 `UrlHelper` 方法，生成实际的 URI。<5>
- 基类为指定的控制器返回一个 `Self` 链接 <6>。派生工厂类可以添加额外的链接，随后将进行介绍。
- `LinkFactory<TController>` 提供更为便捷的强类型使用方式 <7>，不依赖字符串。

7.7.7 IssueLinkFactory

示例 7-8 中的 `IssueLinkFactory`（WebApiBook.IssueTrackerApi\Infrastructure\IssueLink Factory.cs）生成与问题资源相关的所有链接。`IssueLinkFactory` 不包含判断链接是否应该在响应中的逻辑，这部分由 `IssueStateFactory` 处理。

示例 7-8：`IssueLinkFactory` 类

```
public class IssueLinkFactory : LinkFactory<IssueController> // <1>
{
    private const string Prefix = "http://webapibook.net/rels#"; // <5>

    public new class Rels : LinkFactory.Rels { // <3>
        public const string IssueProcessor = Prefix + "issue-processor";
        public const string SearchQuery = Prefix + "search";
```

```
    }

    public class Actions { // <4>
        public const string Open="open";
        public const string Close="close";
        public const string Transition="transition";
    }

    public IssueLinkFactory(HttpRequestMessage request) // <2>
    {
    }

    public Link Transition(string id) // <6>
    {
        return GetLink<IssueProcessorController>(
            Rels.IssueProcessor, id, Actions.Transition);
    }


    public Link Open(string id) { // <7>
        return GetLink<IssueProcessorController>(
            Rels.IssueProcessor, id, Actions.Open);
    }

    public Link Close(string id) { // <8>
        return GetLink<IssueProcessorController>(
            Rels.IssueProcessor, id, Actions.Close);
    }
}
```

`IssueLinkFactory` 类的工作机制如下。

- 这个类派生自 `LinkFactory<IssueController>`，生成指向 `IssueController` 的 self 链接。<1>
- 构造函数参数是一个 `HttpRequestMessage` 实例，这个参数传递给基类。控制器名也传递给基类，用于生成路由。<2>
- 这个工厂类还包含 `Rels`<3> 和 `Actions`<4> 内部类，避免了在调用代码中使用字符串。
- 请注意，`Rel` 前缀 <5> 是指向本书网站的 URI，带有一个 `#`，以获得指定的 `Rel`。
- 这个类还包含 `Transition`<6>、`Open`<7> 和 `Close`<8> 方法，生成用于改变系统状态的链接。

7.8 验收标准

在开始编写 Web API 之前，我们要使用行为驱动测试的 Gherkin 语法，定义大致的验收标准。

下面是 Issue Tracker API 的测试，覆盖了对问题以及问题处理的 CRUD（Create-Read-Update-Delete，增删改查）操作。

```
Feature: Retrieving issues
  Scenario: Retrieving an existing issue
    Given an existing issue
    When it is retrieved
    Then a '200 OK' status is returned
    Then it is returned
    Then it should have an id
    Then it should have a title
    Then it should have a description
    Then it should have a state
    Then it should have a 'self' link
    Then it should have a 'transition' link

  Scenario: Retrieving an open issue
    Given an existing open issue
    When it is retrieved
    Then it should have a 'close' link

  Scenario: Retrieving a closed issue
    Given an existing closed issue
    When it is retrieved
    Then it should have an 'open' link

  Scenario: Retrieving an issue that does not exist
    Given an issue does not exist
    When it is retrieved
    Then a '404 Not Found' status is returned

  Scenario: Retrieving all issues
    Given existing issues
    When all issues are retrieved
    Then a '200 OK' status is returned
    Then all issues are returned
    Then the collection should have a 'self' link

  Scenario: Retrieving all issues as Collection+Json
    Given existing issues
    When all issues are retrieved as Collection+Json
    Then a '200 OK' status is returned
    Then Collection+Json is returned
    Then the href should be set
    Then all issues are returned
    Then the search query is returned

  Scenario: Searching issues
    Given existing issues
        When issues are searched
        Then a '200 OK' status is returned
        Then the collection should have a 'self' link
        Then the matching issues are returned

Feature: Creating issues
  Scenario: Creating a new issue
    Given a new issue
    When a POST request is made
```

```
    Then a '201 Created' status is returned
    Then the issue should be added
    Then the response location header will be set to the resource location

Feature: Updating issues
 Scenario: Updating an issue
    Given an existing issue
    When a PATCH request is made
    Then a '200 OK' is returned
    Then the issue should be updated

  Scenario: Updating an issue that does not exist
    Given an issue does not exist
    When a PATCH request is made
    Then a '404 Not Found' status is returned

Feature: Deleting issues
  Scenario: Deleting an issue
    Give an existing issue
    When a DELETE request is made
    Then a '200 OK' status is returned
    Then the issue should be removed

  Scenario: Deleting an issue that does not exist
    Given an issue does not exist
    When a DELETE request is made
    Then a '404 Not Found' status is returned

Feature: Processing issues
  Scenario: Closing an open issue
    Given an existing open issue
    When a POST request is made to the issue processor
    And the action is 'close'
    Then a '200 OK' status is returned
    Then the issue is closed

  Scenario: Transitioning an open issue
    Given an existing open issue
    When a POST request is made to the issue processor
    And the action is 'transition'
    Then a '200 OK' status is returned
    The issue is closed

  Scenario: Closing a closed issue
    Given an existing closed issue
    When a POST request is made to the issue processor
    And the action is 'close'
    Then a '400 Bad Request' status is returned

  Scenario: Opening a closed issue
    Given an existing closed issue
    When a POST request is made to the issue processor
    And the action is 'open'
    Then a '200 OK' status is returned
    Then it is opened
```

```
Scenario: Transitioning a closed issue
  Given an existing closed issue
  When a POST request is made to the issue processor
  And the action is 'transition'
  Then a '200 OK' status is returned
  Then it is opened

Scenario: Opening an open issue
  Given an existing open issue
  When a POST request is made to the issue processor
  And the action is 'open'
  Then a '400 Bad Request' status is returned

Scenario: Performing an invalid action
  Given an existing issue
  When a POST request is made to the issue processor
  And the action is not valid
  Then a '400 Bad Request' status is returned

Scenario: Opening an issue that does not exist
  Given an issue does not exist
  When a POST request is made to the issue processor
  And the action is 'open'
  Then a '404 Not Found' status is returned

Scenario: Closing an issue that does not exist
  Given an issue does not exist
  When a POST request is made to the issue processor
  And the action is 'close'
  Then a '404 Not Found' status is returned

Scenario: Transitioning an issue that does not exist
  Given an issue does not exist
  When a POST request is made to the issue processor
  And the action is 'transition'
  Then a '404 Not Found' status is returned
```

在这一章之后的部分，你将深入了解获取、创建、更新和删除的全部实现和所有测试。问题处理还有更多的测试，在这里无法全面介绍。但是，我们会介绍控制器 `IssueController` 的测试。你可以在 GitHub 库找到完整的实现和测试代码。

7.9 功能：获取问题

这个功能包括使用一个 HTTP `GET` 方法从 API 获取一个或多个问题。获取问题的响应中包含基于问题状态动态生成的超媒体，因此对应的测试特别完备。

请打开测试文件 RetrievingIssues.cs（WebApiBook.IssueTrackerApi.AcceptanceTests/Features/RetrievingIssues.cs）。请注意，`RetrievingIssues` 派生自示例 7-9 中的 `IssueFeature` 类（参见 IssuesFeature.cs）。这个类是所有测试的基础类，为 API 建立一个内存中（in-memory）

的宿主，测试可以向这个宿主发送 HTTP 请求。

示例 7-9：IssueFeature 类

```
public abstract class IssuesFeature
{
    public Mock<IIssueStore> MockIssueStore;
    public HttpResponseMessage Response;
    public IssueLinkFactory IssueLinks;
    public IssueStateFactory StateFactory;
    public IEnumerable<Issue> FakeIssues;
    public HttpRequestMessage Request { get; private set; }
    public HttpClient Client;

    public IssuesFeature()
    {
        MockIssueStore = new Mock<IIssueStore>(); // <1>
        Request = new HttpRequestMessage();
        Request.Headers.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/vnd.issue+json"));
        IssueLinks = new IssueLinkFactory(Request);
        StateFactory = new IssueStateFactory(IssueLinks);
        FakeIssues = GetFakeIssues(); // <2>
        var config = new HttpConfiguration();
        WebApiConfiguration.Configure(
            config, MockIssueStore.Object);
        var server = new HttpServer(config); // <3>
        Client = new HttpClient(server); // <4>
    }

    private IEnumerable<Issue> GetFakeIssues()
    {
        var fakeIssues = new List<Issue>();
        fakeIssues.Add(new Issue { Id = "1", Title = "An issue",
            Description = "This is an issue",
            Status = IssueStatus.Open });
        fakeIssues.Add(new Issue { Id = "2", Title = "Another issue",
            Description = "This is another issue",
            Status = IssueStatus.Closed });
        return fakeIssues;
    }
}
```

`IssueFeature` 的构造函数为之前提到的服务初始化实例 / 模拟对象，这些操作对所有测试都是通用的：

- 创建一个 `HttpRequest`<1>，准备测试数据 <2>；
- 初始化一个 `HttpServer`，传入使用 `Configure` 方法准备的配置对象 <3>；
- 把 `Client` 属性设置为一个新的 `HttpClient` 实例，在其构造函数中传入 `HttpServer` <4>。

示例 7-10 展示了 `WebApiConfiguration` 类。

示例 7-10：`WebApiConfiguration` 类

```
public static class WebApiConfiguration
{

    public static void Configure(HttpConfiguration config,
    IIssueStore issueStore = null)
    {
        config.Routes.MapHttpRoute("DefaultApi", // <1>
            "{controller}/{id}", new { id = RouteParameter.Optional });
        ConfigureFormatters(config);
        ConfigureAutofac(config, issueStore);
    }

    private static void ConfigureFormatters(HttpConfiguration config)
    {
        config.Formatters.Add(new CollectionJsonFormatter()); // <2>
        JsonSerializerSettings settings = config.Formatters.JsonFormatter.
            SerializerSettings; // <3>
        settings.NullValueHandling = NullValueHandling.Ignore;
        settings.Formatting = Formatting.Indented;
        settings.ContractResolver =
            new CamelCasePropertyNamesContractResolver();
        config.Formatters.JsonFormatter.SupportedMediaTypes.Add(
            new MediaTypeHeaderValue("application/vnd.issue+json"));
    }

    private static void ConfigureAutofac(HttpConfiguration config,
             IIssueStore issueStore)
    {
        var builder = new ContainerBuilder(); // <4>
        builder.RegisterApiControllers(typeof(IssueController).Assembly);

        if (issueStore == null) // <5>
            builder.RegisterType<InMemoryIssueStore>().As<IIssueStore>().
                                 InstancePerLifetimeScope();
        else
            builder.RegisterInstance(issueStore);

        builder.RegisterType<IssueStateFactory>(). // <6>
            As<IStateFactory<Issue, IssueState>>().InstancePerLifetimeScope();
        builder.RegisterType<IssueLinkFactory>().InstancePerLifetimeScope();
        builder.RegisterHttpRequestMessage(config); // <7>
        var container = builder.Build(); // <8>
        config.DependencyResolver = new AutofacWebApiDependencyResolver(container);
    }
}
```

示例 7-10 中的 `WebApiConfiguration.Configure` 方法进行了如下操作：

- 注册默认路由 <1>；
- 添加 `Collection+Json` 格式化程序 <2>；
- 配置默认的 JSON 格式化程序，使其忽略空值，对属性使用骆驼拼写法（camel casing），支持 `Issue` 的媒体类型 <3>。

- 创建一个 Autofac `ContainerBuilder`，注册所有的控制器 <4>；
- 如果传入 issueStore（用于传入模拟实例）<5>，则使用传入的实例注册，否则默认使用 `InMemoryStore`；
- 注册其他服务 <6>；
- 注入当前的 `HttpRequestMessage` 作为依赖 <7>，以使 `IssueLinkFactory` 这样的工作类服务能够接受请求；
- 创建容器，并将其传给 Autofac 依赖关系解析程序 <8>。

7.9.1 获取一个问题

第一组测试验证单个问题的获取以及所有必要的数据如下：

```
Scenario: Retrieving an existing issue
  Given an existing issue
  When it is retrieved
  Then a '200 OK' status is returned
  Then it is returned
  Then it should have an id
  Then it should have a title
  Then it should have a description
  Then it should have a state
  Then it should have a 'self' link
  Then it should have a 'transition' link
```

相关的测试代码在示例 7-11 中。

示例 7-11：获取一个问题

```
[Scenario]
public void RetrievingAnIssue(IssueState issue, Issue fakeIssue)
{
    "Given an existing issue".
        f(() =>
            {
                fakeIssue = FakeIssues.FirstOrDefault();
                MockIssueStore.Setup(i => i.FindAsync("1")).
                    Returns(Task.FromResult(fakeIssue)); // <1>
            });
    "When it is retrieved".
        f(() =>
            {
                Request.RequestUri = _uriIssue1; // <2>
                Response = Client.SendAsync(Request).Result; // <3>
                issue = Response.Content.ReadAsAsync<IssueState>().Result; // <4>
            });
    "Then a '200 OK' status is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <5>
    "Then it is returned".
        f(() => issue.ShouldNotBeNull()); // <6>
    "Then it should have an id".
```

```
            f(() => issue.Id.ShouldEqual(fakeIssue.Id)); // <7>
        "Then it should have a title".
            f(() => issue.Title.ShouldEqual(fakeIssue.Title)); // <8>
        "Then it should have a description".
            f(() => issue.Description.ShouldEqual(fakeIssue.Description)); // <9>
        "Then it should have a state".
            f(() => issue.Status.ShouldEqual(fakeIssue.Status)); // <10>
        "Then it should have a 'self' link".
            f(() =>
                {
                    var link = issue.Links.FirstOrDefault(l => l.Rel ==
                                    IssueLinkFactory.Rels.Self);
                    link.ShouldNotBeNull(); // <11>
                    link.Href.AbsoluteUri.ShouldEqual(
                        "http://localhost/issue/1"); // <12>
                });
        "Then it should have a transition link".
            f(() =>
                {
                    var link = issue.Links.FirstOrDefault(l =>
                        l.Rel == IssueLinkFactory.Rels.IssueProcessor &&
                        l.Action == IssueLinkFactory.Actions.Transition);
                    link.ShouldNotBeNull(); // <13>
                    link.Href.AbsoluteUri.ShouldEqual(
                        "http://localhost/issueprocessor/1?action=transition"); // <14>
                });
    }
```

理解测试

对于不熟悉 XBehave.NET 的人，这里使用的测试语法可能有些令人困惑。在 XBehave 中，一个具体场景的测试组织在一个类方法中，这个方法用 `[Sceanrio]` 属性标记。每个方法可以有一个或多个参数（如 `issue` 和 `fakeIssue`），XBehave 会把这些参数设置为默认值，而不是定义内联变量。

每个方法中有一个或多个待执行的测试。XBehave 允许使用“free form string”语法，直接用英语对测试进行描述。`f()` 函数是 `System.String` 的一个扩展方法，使用 Lambda 表达式。测试中提供的字符串只是为阅读测试代码和 / 或查看结果的用户提供的文档，对 XBehave 自身并没有意义。在实践中，人们使用 Gherkin 语法书写这个字符串，但实际上也不是必须的。XBehave 只关心 Lambda 表达式，按照定义的顺序依次执行表达式。

测试中的另一个常见模式是 `Should` 库的使用。`Should` 库引入了一组以 `Should` 开头的扩展方法，用于执行断言，提供的语法比 `Assert` 方法更加简练。在问题获取功能的测试中，`ShouldEqual` 和 `ShouldNotBeNull` 方法调用都是使用的 `Should` 库。

之前测试的执行概要如下：

- 准备返回一个问题的模拟存储库 <1>；
- 将请求 URI 设置为问题资源 <2>；

- 发送请求 <3>，从响应中获取问题 <4>；
- 验证状态码为 200 <5>；
- 验证获取的问题不为空 <6>；
- 验证获得的 `id` <7>、`title` <8>、`description` <9> 和 `status` <10> 与传入模拟库的问题一致；
- 验证获得问题有 Self 链接，链接指向问题资源；
- 验证获得的问题有 Transition 链接，链接指向问题处理资源。

单个问题的请求由 `IssueController` 的 `Get` 过载方法处理，如示例 7-12 所示。

示例 7-12：`IssueController` 的 `Get` 过载方法

```
public async Task<HttpResponseMessage> Get(string id)
{
    var result = await _store.FindAsync(id); // <1>
    if (result == null)
        return Request.CreateResponse(HttpStatusCode.NotFound); // <2>

    return Request.CreateResponse(HttpStatusCode.OK,
    _stateFactory.Create(result)); // <3>
}
```

这个方法查询一个问题 <1>，如果问题资源未找到则返回一个 `404 Not Found` <2>，如果找到则返回单个资源，而不是层次更高的文档 <3>。

正如你所看到的，这些测试实际上大部分测试的不是控制器自身，而是之前示例 7-6 中的 `IssueStateFactory.Create` 方法。

7.9.2　获取未关闭的和已关闭的问题

```
Scenario: Retrieving an open issue
  Given an existing open issue
  When it is retrieved
  Then it should have a 'close' link

Scenario: Retrieving a closed issue
  Given an existing closed issue
  When it is retrieved
  Then it should have an 'open' link
```

这两个场景测试代码在示例 7-13 和示例 7-14 中。

这一组测试非常类似，一个测试检查一个未关闭问题的 `close` 链接（示例 7-13），另一个测试检查一个已关闭问题的 `open` 链接（示例 7-14）。

示例 7-13：获取一个未关闭问题

```
[Scenario]
public void RetrievingAnOpenIssue(Issue fakeIssue, IssueState issue)
```

```
{
    "Given an existing open issue".
        f(() =>
            {
                fakeIssue = FakeIssues.Single(i =>
                    i.Status == IssueStatus.Open);
                MockIssueStore.Setup(i => i.FindAsync("1")).Returns(
                    Task.FromResult(fakeIssue)); // <1>
            });
    "When it is retrieved".
        f(() =>
            {
                Request.RequestUri = _uriIssue1; // <2>
                issue = Client.SendAsync(Request).Result.Content.
                    ReadAsAsync<IssueState>().Result; // <3>
            });
    "Then it should have a 'close' action link".
        f(() =>
            {
                var link = issue.Links.FirstOrDefault(
                    l => l.Rel == IssueLinkFactory.Rels.IssueProcessor &&
                    l.Action == IssueLinkFactory.Actions.Close); // <4>
                link.ShouldNotBeNull();
                link.Href.AbsoluteUri.ShouldEqual(
                    "http://localhost/issueprocessor/1?action=close");
            });
}
```

示例 7-14：获取一个已关闭问题

```
public void RetrievingAClosedIssue(Issue fakeIssue, IssueState issue)
{
    "Given an existing closed issue".
        f(() =>
            {
                fakeIssue = FakeIssues.Single(i =>
                    i.Status == IssueStatus.Closed);
                MockIssueStore.Setup(i => i.FindAsync("2")).Returns(
                    Task.FromResult(fakeIssue)); // <1>
            });
    "When it is retrieved".
        f(() =>
            {
                Request.RequestUri = _uriIssue2; // <2>
                issue = Client.SendAsync(Request).Result.Content.
                    ReadAsAsync<IssueState>().Result; // <3>
            });
    "Then it should have a 'open' action link".
        f(() =>
            {
                var link = issue.Links.FirstOrDefault(
                    l => l.Rel == IssueLinkFactory.Rels.IssueProcessor &&
                    l.Action == IssueLinkFactory.Actions.Open); // <4>
                link.ShouldNotBeNull();
                link.Href.AbsoluteUri.ShouldEqual(
                    "http://localhost/issueprocessor/2?action=open");
```

```
        });
    }
```

这两个测试的实现也非常相似：

- 准备模拟问题库，返回适用于测试的未关闭问题（`id=1`）或者已关闭问题（`id=2`）<1>；
- 将请求 URI 设置为要获取的资源 <2>；
- 发送请求，获取结果中的资源 <3>；
- 验证相应的 `Open` 或 `CLose` 链接存在 <4>。

与之前的测试类似，这个测试也是验证 `IssueStateFactory` 中的逻辑（参见示例 7-15）。这段逻辑根据问题的状态添加适当的链接。

示例 7-15：`IssueStateFactory Create` 方法

```
public IssueState Create(Issue issue)
{
    ...
    switch (model.Status) {
        case IssueStatus.Closed:
            model.Links.Add(_links.Open(issue.Id));
            break;
        case IssueStatus.Open:
            model.Links.Add(_links.Close(issue.Id));
            break;
    }
    return model;
}
```

7.9.3　获取不存在的问题

下一个场景验证，系统在资源不存在时返回 `404 Not Found`：

```
Scenario: Retrieving an issue that does not exist
  Given an issue does not exist
  When it is retrieved
  Then a '404 Not Found' status is returned
```

示例 7-16 中展示了场景测试代码。

示例 7-16：获取一个不存在的资源

```
[Scenario]
public void RetrievingAnIssueThatDoesNotExist()
{
    "Given an issue does not exist".
        f(() => MockIssueStore.Setup(i =>
            i.FindAsync("1")).Returns(Task.FromResult((Issue)null))); // <1>
    "When it is retrieved".
        f(() =>
            {
```

```
                Request.RequestUri = _uriIssue1; // <2>
                Response = Client.SendAsync(Request).Result; // <3>
            });
    "Then a '404 Not Found' status is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.NotFound)); // <4>
}
```

测试代码工作过程如下。

- 准备返回一个空问题的存储库 <1>。请注意，`Task.FromResult` 扩展用于快速创建一个结果中包含空对象的 `Task`。
- 设置请求 URI <2>。
- 发送请求，获取响应 <3>。
- 验证返回状态码为 `HttpStatusCode.NotFound` <4>。

在 `IssueController.Get` 方法中，这个场景的处理代码如示例 7-17。

示例 7-17：`IssueController.Get` 方法返回一个 404

```
if (result == null)
    return Request.CreateResponse(HttpStatusCode.NotFound);
```

7.9.4 获取所有问题

这个测试场景验证可以正确获取问题集合：

```
Scenario: Retrieving all issues
  Given existing issues
  When all issues are retrieved
  Then a '200 OK' status is returned
  Then all issues are returned
  Then the collection should have a 'self' link
```

示例 7-18 展示了这个场景的测试代码。

示例 7-18：获取所有问题

```
private Uri _uriIssues = new Uri("http://localhost/issue");
private Uri _uriIssue1 = new Uri("http://localhost/issue/1");
private Uri _uriIssue2 = new Uri("http://localhost/issue/2");

[Scenario]
public void RetrievingAllIssues(IssuesState issuesState)
{
    "Given existing issues".
        f(() => MockIssueStore.Setup(i => i.FindAsync()).Returns(
            Task.FromResult(FakeIssues))); // <1>
    "When all issues are retrieved".
        f(() =>
            {
                Request.RequestUri = _uriIssues; // <2>
```

```
                Response = Client.SendAsync(Request).Result; // <3>
                issuesState = Response.Content.
                    ReadAsAsync<IssuesState>().Result; // <4>
            });
        "Then a '200 OK' status is returned".
            f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <5>
        "Then they are returned".
            f(() =>
            {
                issuesState.Issues.FirstOrDefault(i => i.Id == "1").
                    ShouldNotBeNull(); // <6>
                issuesState.Issues.FirstOrDefault(i => i.Id == "2").
                    ShouldNotBeNull();
            });
        "Then the collection should have a 'self' link".
            f(() =>
            {
                var link = issuesState.Links.FirstOrDefault(
                    l => l.Rel == IssueLinkFactory.Rels.Self); // <7>
                link.ShouldNotBeNull();
                link.Href.AbsoluteUri.ShouldEqual("http://localhost/issue");
            });
    }
```

这些测试验证了一个发送到 `/issue` 的请求返回的所有问题。

- 准备返回伪造问题集合的模拟存储库 <1>。
- 设置指向问题资源的请求 URI <2>。
- 发送请求，获取响应 <3>。
- 读取响应内容，将其转换为 `IssueState` 实例 <4>。`ReadAsAsync` 方法使用与 `HttpContent` 实例相关的格式化程序，从响应内容生成一个对象。
- 验证返回的状态是 `OK` <5>。
- 验证返回了正确的问题 <6>。
- 验证返回了 Self 链接 <7>。

在服务器上，问题资源由 IssueController.cs 文件（WebApiBook.IssueTrackerApi/Controllers/IssueController）处理。`IssueController` 依赖于问题库、问题状态工厂以及问题链接工厂（参见示例 7-19）。

示例 7-19：`IssueController` 构造函数

```
public class IssueController : ApiController
{
    private readonly IIssueStore _store;
    private readonly IStateFactory<Issue, IssueState> _stateFactory;
    private readonly IssueLinkFactory _linkFactory;

    public IssueController(IIssueStore store,
        IStateFactory<Issue, IssueState> stateFactory,
        IssueLinkFactory linkFactory)
```

```
        {
            _store = store;
            _stateFactory = stateFactory;
            _linkFactory = linkFactory;
        }
        ...
    }
```

获取所有问题的请求由无参数的 Get 方法处理（参见示例 7-20）。

示例 7-20：IssueController Get 方法

```
    public async Task<HttpResponseMessage> Get()
    {
        var result = await _store.FindAsync(); // <1>
        var issuesState = new IssuesState(); // <2>
        issuesState.Issues = result.Select(i => _stateFactory.Create(i)); // <3>
        issuesState.Links.Add(new Link{
            Href=Request.RequestUri, Rel = LinkFactory.Rels.Self}); // <4>

        return Request.CreateResponse(HttpStatusCode.OK, issuesState); // <5>
    }
```

请注意，这个方法带有修饰符 async，返回 Task<HttpResponseMessage>。在默认情况下，API 控制器操作都是同步的，因此，调用在执行时会阻塞发起调用的线程。对于发出 I/O 调用的操作来说，这可不是好事——这会减少能够处理请求的线程数量。就问题控制器而言，所有的调用都要用到 I/O，因此使用 async 并返回一个 Task 是比较合理的。我们可以使用 await 关键字，等待大量使用 I/O 的操作结束。

这段代码执行的操作如下：

- 首先，发起异步调用，执行问题库的 FindAysnc 方法，获取问题 <1>；
- 创建一个 IssuesState 实例，以保存问题数据 <2>；
- 为问题集合中的每个问题调用状态工厂类的 Create 方法 <3>；
- 通过接收到请求的 URI 添加 Self 链接 <4>；
- 创建响应，传入 IssuesSate 实例以创建响应内容 <5>。

在前一段代码中，我们使用了 Request.CreateResponse 方法返回一个 HttpResponse Message。你可能会问，为什么不直接返回数据模型呢？如果返回一个 HttpResponse Message，客户端可以直接使用 HttpResponse 的组件，如状态码和标头。虽然控制器的这个操作现在并没有修改响应标头，但是将来可能会发生这种情况。你还将看到，控制器的其他操作的确会修改响应组件。

应该在哪里处理超媒体？

我们经常会碰到这个问题：应该在系统的什么地方实现超媒体控制？超媒体应该在控制器中处理，还是应该通过管道使用消息处理程序、过滤器或者格式化程序处理？这个问题并没有唯一正确的答案——所有这些地方都可以处理超媒体——但是需要权衡利弊：

- 如果在控制器中处理链接，那么处理过程较为明确 / 清晰，更容易进行单步调试；
- 如果在管道中处理链接，那么控制器行为会更加简洁，包含逻辑较少；
- 消息处理程序、过滤器和控制器可以很方便地访问请求，使用请求信息，生成链接。

在这本书中，我们选择在控制器中处理超媒体，要么采取 Get 方法中获取多个问题时的处理方式，在行内生成链接；要么采取 Get 方法获取单个问题时的方式，使用注入的服务生成链接。这么做的原因是，链接逻辑对控制器更明确 / 接近。控制器的任务是在业务域与 HTTP 世界之间进行转换。因为链接是 HTTP 相关的关注点，因此在控制器中处理链接十分合乎情理。

虽说如此，但其他的链接处理方式也是可行的，并不是不能使用。

7.9.5　获取所有问题的Collection+Json表示

正如前面一章中介绍的，`Collection+Json` 格式非常适合数据列表的管理和查询。问题资源对返回多个条目的资源请求支持 `Collection+Json` 格式的响应。这个测试验证系统可以返回 `Collection+Json` 格式的响应。

下一个测试场景验证 API 正确处理了返回 `Collection+Json` 的请求：

```
Scenario: Retrieving all issues as Collection+Json
  Given existing issues
  When all issues are retrieved as Collection+Json
  Then a '200 OK' status is returned
  Then Collection+Json is returned
  Then the href should be set
  Then all issues are returned
  Then the search query is returned
```

示例 7-21 中的测试发送场景中所描述的请求，并验证返回的格式正确。

示例 7-21：获取所有问题的 `Collection+Json` 表示

```
[Scenario]
public void RetrievingAllIssuesAsCollectionJson(IReadDocument readDocument)
{
    "Given existing issues".
        f(() => MockIssueStore.Setup(i => i.FindAsync()).
```

```
            Returns(Task.FromResult(FakeIssues)));
    "When all issues are retrieved as Collection+Json".
        f(() =>
            {
                Request.RequestUri = _uriIssues;
                Request.Headers.Accept.Clear(); // <1>
                Request.Headers.Accept.Add(
                    new MediaTypeWithQualityHeaderValue(
                        "application/vnd.collection+json"));
                Response = Client.SendAsync(Request).Result;
                readDocument = Response.Content.ReadAsAsync<ReadDocument>(
                    new[] {new CollectionJsonFormatter()}).Result; // <2>
            });
    "Then a '200 OK' status is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <3>
    "Then Collection+Json is returned".
        f(() => readDocument.ShouldNotBeNull()); // <4>
    "Then the href should be set".
        f(() => readDocument.Collection.Href.AbsoluteUri.ShouldEqual(
            "http://localhost/issue")); // <5>
    "Then all issues are returned"
        f(() =>
            {
                readDocument.Collection.Items.FirstOrDefault(
                    i=>i.Href.AbsoluteUri=="http://localhost/issue/1").
                                    ShouldNotBeNull(); // <6>
                readDocument.Collection.Items.FirstOrDefault(
                    i=>i.Href.AbsoluteUri=="http://localhost/issue/2").
                                    ShouldNotBeNull();
            });
    "Then the search query is returned".
        f(() => readDocument.Collection.Queries.SingleOrDefault(
            q => q.Rel == IssueLinkFactory.Rels.SearchQuery).
            ShouldNotBeNull()); // <7>
}
```

在标准的准备操作之后，测试进行了如下操作：

- 将 Accept 标头设置为 applciation/vnd.collection+json，并发送请求 <1>；
- 使用 CollectionJson 软件包的 ReadDocument 读取响应内容 <2>；
- 验证返回的状态码为 200 OK <3>；
- 验证返回的文档不为空（即返回了有效的 Collection+Json）<4>；
- 验证返回文档的 href(self) URI 设置正确 <5>；
- 验证文档中存在预期的条目 <6>；
- 验证文档的 Queries 集合中存在搜索查询 <7>。

在服务器端，系统调用了和前一个测试同样的方法——即 IssueController.Get()。但是，因为我们配置使用了 CollectionJsonFormatter，返回的 IssuesState 对象是通过实现的 IReadDocument 接口输出的（参见示例 7-4）。

7.9.6 搜索问题

这个测试场景验证 API 允许用户执行搜索而且返回结果：

```
Scenario: Searching issues
  Given existing issues
      When issues are searched
      Then a '200 OK' status is returned
      Then the collection should have a 'self' link
      Then the matching issues are returned
```

示例 7-22 展示了这个场景的测试代码。

示例 7-22：搜索问题

```
[Scenario]
public void SearchingIssues(IssuesState issuesState)
{
    "Given existing issues".
        f(() => MockIssueStore.Setup(i => i.FindAsyncQuery("another"))
            .Returns(Task.FromResult(FakeIssues.Where(i=>i.Id == "2")))); // <1>
    "When issues are searched".
        f(() =>
        {
            Request.RequestUri = new Uri(_uriIssues, "?searchtext=another");
            Response = Client.SendAsync(Request).Result;
            issuesState = Response.Content.ReadAsAsync<IssuesState>().Result; // <2>
        });
    "Then a '200 OK' status is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <3>
   "Then the collection should have a 'self' link".
        f(() =>
        {
            var link = issuesState.Links.FirstOrDefault(
                l => l.Rel == IssueLinkFactory.Rels.Self); // <4>
            link.ShouldNotBeNull();
            link.Href.AbsoluteUri.ShouldEqual(
                "http://localhost/issue?searchtext=another");
        });
    "Then the matching issues are returned".
        f(() =>
            {
                var issue = issuesState.Issues.FirstOrDefault(); // <5>
                issue.ShouldNotBeNull();
                issue.Id.ShouldEqual("2");
            });
}
```

这些测试的工作过程如下：

- 设置模拟存储库，使其在调用 `FindAsyncQuery` 时返回问题 2 <1>；
- 在查询 URI 后附加查询字符串，发送请求，将响应内容读取为 `IssueState` 实例 <2>；
- 验证返回状态码为 `200 OK` <3>；

- 验证集合的 Self 链接设置正确 <4>；
- 验证返回了预期的问题 <5>。

示例 7-23 展示了搜索功能的实现代码。

示例 7-23：IssueController 的 GetSearch 方法

```
public async Task<HttpResponseMessage> GetSearch(string searchText) // <1>
{
    var issues = await _store.FindAsyncQuery(searchText); // <2>
    var issuesState = new IssuesState();
    issuesState.Issues = issues.Select(i => _stateFactory.Create(i)); // <3>
    issuesState.Links.Add( new Link {
        Href = Request.RequestUri, Rel = LinkFactory.Rels.Self }); // <4>
    return Request.CreateResponse(HttpStatusCode.OK, issuesState); // <5>
}
```

这段代码的工作方式如下。

- 方法名为 `GetSearch` <1>。ASP.NET Web API 的选择器直接将当前的 HTTP 方法与以同一 HTTP 方法名开头的控制器方法进行匹配。因此，`GetSearch` 方法就会与 HTTP 的 `GET` 方法匹配，查询字符串参数 `searchtext` 与 HTTP 方法的参数匹配。
- `FindAsyncQuery` 方法获取符合查询条件的问题 <2>。
- 创建一个 `IssuesState` 实例，将搜索结果填充其中 <3>。
- 添加一个 Self 链接，指向初始请求 <4>。
- 返回一个有效载荷为搜索到的问题的 OK 响应 <5>。



与获取所有问题的请求类似，这个资源也支持返回 `Collection+Json` 格式的表示。

到这里，获取问题功能的所有场景都介绍完了。现在，接着讨论创建问题功能！

7.10　功能：创建问题

这个功能包括一个场景，即客户端使用 HTTP `POST` 创建一个新问题：

```
Scenario: Creating a new issue
  Given a new issue
  When a POST request is made
  Then the issue should be added
  Then a '201 Created' status is returned
  Then the response location header will be set to the new resource location
```

示例 7-24 展示了这个场景的测试代码。

示例 7-24：创建问题

```
[Scenario]
public void CreatingANewIssue(dynamic newIssue)
{
    "Given a new issue".
        f(() =>
            {
                newIssue = new JObject();
                newIssue.description = "A new issue";
                newIssue.title = "NewIssue"; // <1>
                MockIssueStore.Setup(i => i.CreateAsync(It.IsAny<Issue>())).
                    Returns<Issue>(issue=>
                        {
                            issue.Id = "1";
                            return Task.FromResult("");
                        }); // <2>
            });
    "When a POST request is made".
        f(() =>
            {
                Request.Method = HttpMethod.Post;
                Request.RequestUri = _issues;
                Request.Content = new ObjectContent<dynamic>(
                    newIssue, new JsonMediaTypeFormatter()); // <3>
                Response = Client.SendAsync(Request).Result;
            });
    "Then the issue should be added".
        f(() => MockIssueStore.Verify(i => i.CreateAsync(
        It.IsAny<Issue>()))); // <4>
    "Then a '201 Created' status is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.Created)); // <5>
    "Then the response location header will be set to the resource location".
        f(() => Response.Headers.Location.AbsoluteUri.ShouldEqual(
            "http://localhost/issue/1")); // <6>
}
```

测试代码的工作过程如下。

- 创建一个要发送给服务器的新问题 <1>。
- 配置模拟存储库，设置问题 `Id` <2>，请注意其中的 `Task.FromResult` 调用。`CreateAsync` 方法预期返回一个 `Task`。这是一种创建“假”任务的简单方法。你将看到，如果模拟存储库的方法需要返回一个 `Task` 时，别的测试使用的也是同样的方法。
- 配置请求，设为其为 `POST`，请求内容为新问题 <3>。这里需要注意的是，代码中没有使用静态的 CLR 类型（如 `Issue`），而是用 `JObject` 实例（Json.NET 中定义）转换成 `dynamic`。你稍后就将看到，我们可以在服务器上也使用类似的方法保持无类型。
- 验证调用了 `CreateAsync` 方法创建问题 <4>。
- 验证返回的状态码为 `201`，与 HTTP 规范一致（参见第 1 章）<5>。
- 验证 `Location` 标头设置为新创建资源的地址 <6>。

Web 是无类型的

在示例 7-24 中，客户端创建了一个动态类型发送给服务器，而不是使用静态类型。这可能使得在座的很多人发问："为什么不使用静态类型呢？"的确，.NET 是一个（大体上）静态的语言。但是，Web 是没有类型的。正如我们在第 1 章中介绍的，Web 是基于消息的，而非基于类型。客户端发送给服务器的消息使用一组已知的格式（即媒体类型）。一个媒体类型描述了一个消息的结构，与编程环境（如 .NET）中的静态类型不同。这种无类型的特点并不是偶然产生的，是有意这样设计的。

Web 没有类型，因而可以让尽可能多的客户端和服务器访问资源。也正是因为无类型的特性，客户端和服务器可以独立演化，多个版本并存。服务器可以理解所收到消息中的新元素，已有的客户端并不需要升级。消息格式的演化甚至可以是破坏性的，但仍然无需强制客户端升级。在强类型的世界里，由类型导致的内在约束决定了这些都是不可能的。

SOAP 服务是一个协议将类型引入 Web 的例子，带来了诸多问题。在实际应用中，实现 SOAP 服务的公司最苦恼的是客户端。与 SOAP 服务通信需要使用一个 WSDL 文档，描述操作和类型。如果服务发生了重大的变化，这些变化通常会导致客户端无法使用。解决的办法要么是升级客户端，要么是服务的新版本与旧版本并存，而新客户端只有得到 WSDL 才能使用新版本的服务。

虽说如此，但是我们并不是建议你不使用类型。对于开发者，使用类型是访问 API 请求和响应的更合理的方式，但是这些类型不应该成为与系统交互的必要条件，而且使用动态类型也是完全合理的（实际上，有时使用动态类型比静态类型好处更多）。

示例 7-25 展示了控制器内的实现。

示例 7-25：`IssueController` 的 `Post` 方法

```
public async Task<HttpResponseMessage> Post(dynamic newIssue) // <1>
{
    var issue = new Issue {
        Title = newIssue.title, Description = newIssue.description}; // <2>
    await _store.CreateAsync(issue); // <3>
    var response = Request.CreateResponse(HttpStatusCode.Created); // <4>
    response.Headers.Location = _linkFactory.Self(issue.Id).Href; // <5>
    return response; // <6>.
}
```

代码工作方式如下。

- 方法自身命名为 `POST`，以匹配 HTTP 的 `POST` 方法 <1>。与测试客户端一样，这个方法的参数是 `dynamic` 类型。在服务器上，Json.NET 遇到 `dynamic` 会自动创建一个 `JObject` 实例。虽然默认支持 JSON，我们也可以添加定制的格式化程序，用于支持其他的媒体

类型，如 application/x-www-form-urlencoded。

- 我们传入动态实例的属性，创建一个新问题 <2>。
- 调用存储库的 CreateAsync 方法，存储这个问题 <3>。
- 创建响应，返回 201 Created 状态 <4>。
- 我们调用 _linkFactory 的 Self 方法，设置响应的 location 标头，然后返回响应 <6>。

问题创建功能就介绍完了。接下来，接着介绍问题更新！

7.11　功能：更新问题

这个功能覆盖了使用 HTTP 的 PATCH 更新问题。我们选择 PATCH 方法，因为它允许客户端只发送修改现有资源的部分数据。而 PUT 方法则需要完全替换资源的状态。

7.11.1　更新一个问题

这一场景验证的是，客户端发送一个 PATCH 请求时，相应的资源得到更新：

```
Scenario: Updating an issue
  Given an existing issue
  When a PATCH request is made
  Then a '200 OK' is returned
  Then the issue should be updated
```

示例 7-26 展示了这一场景的测试代码。

示例 7-26：IssueController 的 PATCH 方法

```
[Scenario]
public void UpdatingAnIssue(Issue fakeIssue)
{
    "Given an existing issue".
        f(() =>
            {
                fakeIssue = FakeIssues.FirstOrDefault();
                MockIssueStore.Setup(i => i.FindAsync("1")).Returns(
                    Task.FromResult(fakeIssue)); // <1>
                MockIssueStore.Setup(i => i.UpdateAsync(It.IsAny<Issue>())).
                    Returns(Task.FromResult(""));
            });
    "When a PATCH request is made".
        f(() =>
            {
                dynamic issue = new JObject(); // <2>
                issue.description = "Updated description";
                Request.Method = new HttpMethod("PATCH"); // <3>
                Request.RequestUri = _uriIssue1;
                Request.Content = new ObjectContent<dynamic>(issue,
                    new JsonMediaTypeFormatter()); // <4>
                Response = Client.SendAsync(Request).Result;
```

```
            });
        "Then a '200 OK' status is returned".
            f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <5>
        "Then the issue should be updated".
            f(() => MockIssueStore.Verify(i =>
                i.UpdateAsync(It.IsAny<Issue>()))); // <6>
        "Then the descripton should be updated".
            f(() => fakeIssue.Description.ShouldEqual("Updated description")); // <7>
        "Then the title should not change".
            f(() => fakeIssue.Title.ShouldEqual(title)); // <8>
    }
```

测试的工作过程如下。

- 准备模拟存储库，使其在 FindAsync 调用时返回需要更新的预期问题，并处理 UpdateAsync 调用 <1>。
- 新建一个 JObject 实例，只设置需要修改的描述 <2>。
- 将请求方法设置为 PATCH <3>。请注意，这里创建了一个 HttpMethod 实例，传入的构造函数参数是方法名。使用一个 HTTP 方法时，如果不预设 HttpMethod 类的静态属性（如 GET、PUT、POST 和 DELETE），就可以采取这种构造方式。
- 使用已创建的问题，新建一个 ObjectContent<dynamic> 实例，将其赋给请求内容。然后发送请求 <4>。请注意 dynamic 的用法：动态类型非常适合 PATCH 方法，客户端可以只发送需要更新的问题属性。
- 验证返回的状态码是 200 OK <5>。
- 验证调用了 UpdateAsync 方法，传入参数是待更新的问题 <6>。
- 验证问题的描述已更新 <7>。
- 验证问题的标题未改变 <8>。

问题更新的实现由控制器中的 Patch 方法处理，其代码如示例 7-27 所示。

示例 7-27：IssueController 的 Patch 方法

```
    public async Task<HttpResponseMessage> Patch(string id, dynamic issueUpdate) // <1>
    {
        var issue = await _store.FindAsync(id); // <2>
        if (issue == null) // <3>
            return Request.CreateResponse(HttpStatusCode.NotFound);

        foreach (JProperty prop in issueUpdate) // <4>
        {
            if (prop.Name == "title")
                issue.Title = prop.Value.ToObject<string>();
            else if (prop.Name == "description")
                issue.Description = prop.Value.ToObject<string>();
        }
        await _store.UpdateAsync(issue); // <5>
        return Request.CreateResponse(HttpStatusCode.OK); // <6>
    }
```

代码工作方式如下。

- 这个方法使用两个参数 <1>。参数 id 来自请求的 URI（在我们的例子中，值为 http://localhost/issue/1），参数 issueUpdate 来自请求的 JSON 内容。
- 从存储库中获取待更新的问题 <2>。
- 如果待更新问题没有找到，那么立即返回一个 404 Not Found <3>。
- 循环遍历 issueUpdate 的属性，只更新那些存在的属性 <4>。
- 调用存储库，更新问题 <5>。
- 返回一个 200 OK 状态 <6>。

7.11.2 更新不存在的问题

这个场景确保，当客户端向一个缺失或已删除的问题发送一个 PATCH 请求时，系统返回一个 404 Not Found 状态：

```
Scenario: Updating an issue that does not exist
  Given an issue does not exist
  When a PATCH request is made
  Then a '404 Not Found' status is returned
```

在前一节中我们已经讨论了控制器中的相关代码，而示例 7-28 中的测试验证了代码逻辑正确，运行无误！

示例 7-28：更新一个不存在的问题

```
[Scenario]
public void UpdatingAnIssueThatDoesNotExist()
{
    "Given an issue does not exist".
        f(() => MockIssueStore.Setup(i => i.FindAsync("1")).
                Returns(Task.FromResult((Issue)null))); // <1>
    "When a PATCH request is made".
        f(() =>
            {
                Request.Method = new HttpMethod("PATCH"); // <2>
                Request.RequestUri = _uriIssue1;
                Request.Content = new ObjectContent<dynamic>(new JObject(),
                    new JsonMediaTypeFormatter()); // <3>
                response = Client.SendAsync(Request).Result; // <4>
            });
    "Then a 404 Not Found status is returned".
        f(() => response.StatusCode.ShouldEqual(HttpStatusCode.NotFound)); // <5>
}
```

测试的工作过程如下：

- 准备模拟存储库，使其在 FindAsync 调用时返回一个空问题；
- 将请求方法设置为 PATCH <2>；

- 将请求内容设置为一个空 JObject 实例，请求内容是什么其实无关紧要 <3>；
- 发送请求 <4>；
- 验证返回状态为 404 Not Found。

到这里，更新问题部分就介绍完了。

7.12 功能：删除问题

这个功能负责处理删除问题的 HTTP DELETE 请求。

7.12.1 删除一个问题

这个场景验证了，当客户端发送一个 DELETE 请求时，相应的问题得到移除：

```
Scenario: Deleting an issue
  Give an existing issue
  When a DELETE request is made
  Then a '200 OK' status is returned
  Then the issue should be removed
```

这个场景的测试（示例 7-29）相当简单，这一章前面的内容已经介绍了测试中使用到的概念。

示例 7-29：删除一个问题

```
[Scenario]
public void DeletingAnIssue(Issue fakeIssue)
{
    "Given an existing issue".
        f(() =>
            {
                fakeIssue = FakeIssues.FirstOrDefault();
                MockIssueStore.Setup(i => i.FindAsync("1")).Returns(
                    Task.FromResult(fakeIssue)); // <1>
                MockIssueStore.Setup(i => i.DeleteAsync("1")).Returns(
                    Task.FromResult(""));
            });
    "When a DELETE request is made".
        f(() =>
            {
                Request.RequestUri = _uriIssue;
                Request.Method = HttpMethod.Delete; // <2>
                Response = Client.SendAsync(Request).Result; // <3>
            });
    "Then the issue should be removed".
        f(() => MockIssueStore.Verify(i => i.DeleteAsync("1"))); // <4>
    "Then a '200 OK status' is returned".
        f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <5>
}
```

测试工作过程如下：

- 配置存储库，使其在 FindAsync 调用时返回待删除的问题，并处理 DeleteAsync 调用 <1>；
- 设置请求使用 DELETE 方法 <2>，并发送请求 <3>；
- 验证调用了 DeleteAsync 方法，传入参数为 Id <4>；
- 验证响应状态为 200 OK <5>。

示例 7-30 展示了控制器中的实现代码。

示例 7-30：IssueController 的 Delete 方法

```
public async Task<HttpResponseMessage> Delete(string id) // <1>
{
    var issue = await _store.FindAsync(id); // <2>
    if (issue == null)
        return Request.CreateResponse(HttpStatusCode.NotFound); // <3>
    await _store.DeleteAsync(id); // <4>
    return Request.CreateResponse(HttpStatusCode.OK); // <5>
}
```

这段代码进行了如下工作。

- 方法名为 Delete，以匹配 HTTP 的 DELETE 方法 <1>。方法参数为待删除问题的 id。
- 从存储库获取所选 id 的问题 <2>。
- 如果该问题不存在，那么返回一个 404 Not Found 状态 <3>。
- 调用存储库的 DeleteAsync 方法，删除指定的问题 <4>。
- 向客户端返回一个 200 OK <5>。

7.12.2　删除不存在的问题

这个场景验证的是，如果客户端向一个不存在的问题发送一个 DELETE 请求，那么系统返回以 404 Not Found 状态：

```
Scenario: Deleting an issue that does not exist
  Given an issue does not exist
  When a DELETE request is made
  Then a '404 Not Found' status is returned
```

前面我们讨论过更新一个不存在问题的测试，示例 7-31 中的测试与其非常相似。

示例 7-31：删除一个不存在的问题

```
[Scenario]
public void DeletingAnIssueThatDoesNotExist()
{
    "Given an issue does not exist".
        f(() => MockIssueStore.Setup(i => i.FindAsync("1")).Returns(
            Task.FromResult((Issue) null))); // <1>
```

```
        "When a DELETE request is made".
            f(() =>
                {
                    equest.RequestUri = _uriIssue;
                    Request.Method = HttpMethod.Delete; // <2>
                    Response = Client.SendAsync(Request).Result;
                });
        "Then a '404 Not Found' status is returned".
            f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.NotFound)); // <3>
    }
```

测试工作过程如下：

- 设置模拟存储库，使其在请求问题时返回空值 <1>；
- 发送删除资源的请求 <2>；
- 验证返回状态为 `404 Not Found` <3>。

7.13　功能：处理问题

7.13.1　测试

正如前面提到的，这个功能的测试讨论超出了这一章的讲述范围。但是，你现在已经掌握了理解相关代码（可从 GitHub 下载：https://github.com/webapibook/issuetracker/blob/BuildingTheApi/test/WebApiBook.IssueTrackerApi.AcceptanceTests/Features/ProcessingIssues.cs）所需的所有概念。

将资源的处理独立出去，可以更好地隔离 API 的实现，提高代码的可读性和更易于维护性。采用这种设计方式，你可以在不改动 `IssueController` 的情况下修改资源处理的逻辑，符合单一责任原则（Single Responsibility Principle），提高了系统的可演化性。

7.13.2　实现

问题处理资源由 `IssueProcessorController`（代码见示例 7-32）实现。

示例 7-32：`IssueProcessorController`

```
    public class IssueProcessorController : ApiController
    {
        private readonly IIssueStore _issueStore;

        public IssueProcessorController(IIssueStore issueStore)
        {
            _issueStore = issueStore; // <1>
        }

        public async Task<HttpResponseMessage> Post(string id, string action) // <2>
        {
```

```
            bool isValid = IsValidAction(action); // <3>
            Issue issue = null;

            if (isValid)
            {
                issue = await _issueStore.FindAsync(id); // <4>

                if (issue == null)
                    return Request.CreateResponse(HttpStatusCode.NotFound); // <5>

                if ((action == IssueLinkFactory.Actions.Open ||
                    action == IssueLinkFactory.Actions.Transition) &&
                    issue.Status == IssueStatus.Closed)
                {
                    issue.Status = IssueStatus.Open; // <6>
                }
                else if ((action == IssueLinkFactory.Actions.Close ||
                    action == IssueLinkFactory.Actions.Transition) &&
                    issue.Status == IssueStatus.Open)
                {
                    issue.Status = IssueStatus.Closed; // <7>
                }
                else
                    isValid = false; // <8>
            }

            if (!isValid)
                return Request.CreateErrorResponse(HttpStatusCode.BadRequest,
                    string.Format("Action '{0}' is invalid", action)); // <9>

            await _issueStore.UpdateAsync(issue); // <10>
            return Request.CreateResponse(HttpStatusCode.OK); // <11>
        }

        public bool IsValidAction(string action)
        {
            return (action == IssueLinkFactory.Actions.Close ||
                action == IssueLinkFactory.Actions.Open ||
                action == IssueLinkFactory.Actions.Transition);
        }
    }
```

代码工作方式如下。

- `IssueProcessorController` 的构造函数参数为 `IIssueStore`，与 `IssueController` 类似 <1>。
- 控制器方法为 `Post`，接受来自请求 URI 的 `id` 和 `action` <2>。
- 调用 `IsValidAction` 方法，检查是否能够识别请求的操作 <3>。
- 调用 `FindAsync` 方法，获取问题 <4>。
- 如果问题没有找到，那么立即返回一个 `400 Not Found` <5>。
- 如果操作为 `open` 或 `transition`，而且问题已关闭，那么打开问题 <6>。
- 如果操作为 `close` 或 `transition`，而且问题未关闭，那么关闭问题 <7>。

- 如果前两个条件都不满足，那么标识该操作为对当前状态无效 <8>。
- 如果操作无效，那么通过 `CreateErrorResponse` 返回一个错误。我们使用这个方法，是为了使错误响应包含有效负载 <9>。
- 调用 `UpdateAsync` 方法更新问题 <10>，返回一个 `200 OK` 状态 <11>。

到这里，Issue Tracker API 就全部介绍完毕了！

7.14　小结

这一章的内容相当丰富。我们从系统的概要设计一直讨论到 API 的详细需求及其具体实现。在这一过程中，我们了解了 Web API 在实践中需要考虑的很多方面，以及如何使用内存托管进行集成测试。这些概念对于使用 ASP.NET 构建可演化的 API 非常重要。现在我们可以开始讨论有趣的东西了！在下一章，你将看到如何改进这个 API，了解使 API 可扩展的必要工具（如缓存）。

第8章

改进API

付出才有回报，敢于尝试才能成功。

在前一章中，我们讨论了问题跟踪系统的初始实现。有了这个完整的可运行的实现，我们就可以用它来讨论 API 的设计以及支持 API 的媒体类型。在本章中，我们将试图改进现有的实现，添加一些新功能，如缓存、冲突检测和安全性。我们依然会采取在初始实现中的做法，用行为驱动测试的方式描述这些新功能的所有需求。在添加这些新功能时，我们将深入实现的细节，阅读真实代码，并介绍其背后的理论。随后的章节将会更为详细地讨论这些理论。

8.1 新功能的验收标准

以下是我们的 API 测试场景，定义了问题跟踪系统的新需求：

```
Feature: Output Caching
  Scenario: Retrieving existing issues
    Given existing issues
    When all issues are retrieved
    Then a CacheControl header is returned
    Then a '200 OK' status is returned
    Then all issues are returned

  Scenario: Retrieving an existing issue
    Given an existing issue
    When it is retrieved
    Then a LastModified header is returned
    Then a CacheControl header is returned
    Then a '200 OK' status is returned
```

```
    Then it is returned

Feature: Cache revalidation

  Scenario: Retrieving an existing issue that has not changed
    Given an existing issue
    When it is retrieved with an IfModifiedSince header
    Then a CacheControl header is returned
    Then a '304 NOT MODIFIED' status is returned
    Then it is returned

  Scenario: Retrieving an existing issue that has changed
    Given an existing issue
    When it is retrieved with an IfModifiedSince header
    Then a LastModified header is returned
    Then a CacheControl header is returned
    Then a '200 OK' status is returned
    Then it is returned

Feature: Conflict detection

  Scenario: Updating an issue with no conflicts
    Given an existing issue
    When a PATCH request is made with an IfModifiedSince header
    Then a '200 OK' is returned
    Then the issue should be updated

  Scenario: Updating an issue with conflicts
    Given an existing issue
    When a PATCH request is made with an IfModifiedSince header
    Then a '409 CONFLICT' is returned
    Then the issue is not updated

Feature: Change Auditing

  Scenario: Creating a new issue
    Given a new issue
    When a POST request is made with an Authorization header containing the user
        identifier
    Then a '201 Created' status is returned
    Then the issue should be added with auditing information
    Then the response location header will be set to the resource location

  Scenario: Updating an issue
    Given an existing issue
    When a PATCH request is made with an Authorization header containing the user
        identifier
    Then a '200 OK' is returned
    Then the issue should be updated with auditing information

Feature: Tracing

  Scenario: Creating, Updating, Deleting, or Retrieving an issue
    Given an existing or new issue
    When a request is made
```

```
When the diagnostics tracing is enabled
Then the diagnostics tracing information is generated
```

8.2 实现输出缓存支持

缓存是使互联网可扩展的基础功能之一，如果正确实施，可以提供如下的优点。

- 降低源服务器的负载。
- 减少网络延迟。客户端可以更快得到响应。
- 节省网络带宽。在请求到达源服务器之前，所需的内容就可能在某些缓存中间层中找到，因而减少网络跳转。

在 Web API 上正确实现缓存机制，主要需要两个步骤：

(1) 设置正确的标头，告诉中间层和客户端（例如：代理、反向代理、本地缓存、浏览器等）对响应进行缓存；
(2) 实现条件 `GET`，使中间层能够在缓存的副本过期后重新验证生效。

第 (1) 步需要使用 `Expire` 或 `Cache-Control` 标头。`Expire` 标头可以用于设置绝对过期时间，告诉缓存相关的表示多长时间有效。大部分 API 实现使用这个标头说明客户端最后一次获取这个表示的时间，或者服务器上文档最后一次修改的时间。这个标头的值必须使用 GTM 时间表示，而不是本地时间——例如：`Expires: Mon, 1 Aug 2013 10:30:50 GMT`。另一方面，`Cache-Control` 标头提供更为精细的控制，说明相对的过期时间，以及谁可以缓存数据。下面的列表描述了 `Cache-Control` 标头的常用值。

- `no-store`

  说明缓存在任何情况下都不应该保存数据的副本。

- `private`

  说明数据只供一个用户使用，因此应该保存在私有缓存（如浏览器缓存）中，而不能保存在共享缓存（如代理缓存）中。

- `public`

  说明数据可以在任意地方进行缓存。

- `no-cache`

  强制缓存在已缓存副本过期后重新进行验证。

- `max-age`

  说明一个以秒为单位的时间增量，表示一个缓存副本保持有效的最长时间（例如：`max-age[300]` 表示缓存副本将在请求发出 300 秒后过期）。

- `s-maxage`

  等同于 `max-age`，但是只对共享缓存有效。

8.3 添加输出缓存测试

我们要做的第一件事就是添加一个新文件 OutputCaching，用于所有与输出缓存相关的测试。我们的第一个测试是，在返回所有问题的操作中加入输出缓存支持：

```
Scenario: Retrieving existing issues
    Given existing issues
    When all issues are retrieved
    Then a CacheControl header is returned
    Then a '200 OK' status is returned
    Then all issues are returned
```

我们使用行为驱动开发，将这个场景转换成一个单元测试（参见示例 8-1）。

示例 8-1：获取带有缓存标头的所有问题

```
public class OutputCaching : IssuesFeature
{
  private Uri _uriIssues = new Uri("http://localhost/issue");

  [Scenario]
  public void RetrievingAllIssues()
  {
    IssuesState issuesState = null;

    "Given existing issues".
      f(() =>
      {
        MockIssueStore.Setup(i => i.FindAsync())
          .Returns(Task.FromResult(FakeIssues))
      });
    "When all issues are retrieved".
      f(() =>
      {
        Request.RequestUri = _uriIssues;
        Response = Client.SendAsync(Request).Result;
        issuesState = Response.Content
          .ReadAsAsync<IssuesState>()
          .Result;
      });
    "Then a CacheControl header is returned".
      f(() =>
      {
        Response.Headers.CacheControl.Public
          .ShouldBeTrue(); // <1>
        Response.Headers.CacheControl.MaxAge
          .ShouldEqual(TimeSpan.FromMinutes(5)); // <2>
      });
    "Then a '200 OK' status is returned".
```

```
      f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK));
    "Then they are returned".
      f(() =>
      {
        issuesState.Issues
          .FirstOrDefault(i => i.Id == "1")
          .ShouldNotBeNull();
        issuesState.Issues
          .FirstOrDefault(i => i.Id == "2")
          .ShouldNotBeNull();
      });
  }
}
```

这个单元测试是自解释的，不用多加说明。关键代码在 <1> 和 <2>，进行 CacheControl 和 MaxAge 断言的地方。要使这个测试通过，IssueController 类的 Get 方法所返回的响应消息需要进行修改，包含 CacheControl 和 MaxAge 标头（参见示例 8-2）。

示例 8-2：Get 方法的新版本

```
public async Task<HttpResponseMessage> Get()
{
  var result = await _store.FindAsync();
  var issuesState = new IssuesState();
  issuesState.Issues = result.Select(i => _stateFactory.Create(i));

  var response = Request.CreateResponse(HttpStatusCode.OK, issuesState);

  response.Headers.CacheControl = new CacheControlHeaderValue();
  response.Headers.CacheControl.Public = true; // <1>
  response.Headers.CacheControl.MaxAge = TimeSpan.FromMinutes(5); // <2>

  return response;
}
```

CacheControl 标头值设置为 Public <1>，因此这个响应可以在任何地方缓存，MaxAge 标头设置为 5 分钟的相对过期时间 <2>。

示例 8-3 中展示的下一个场景，是为获取单个问题的操作添加输出缓存：

```
Scenario: Retrieving an existing issue
Given an existing issue
When it is retrieved
Then a LastModified header is returned
Then a CacheControl header is returned
Then a '200 OK' status is returned
Then it is returned
```

示例 8-3：获取带有缓存标头的单个问题

```
public class OutputCaching : IssuesFeature
{
  private Uri _uriIssue1 = new Uri("http://localhost/issue/1");
```

```
    [Scenario]
    public void RetrievingAnIssue()
    {
      IssueState issue = null;

       var fakeIssue = FakeIssues.FirstOrDefault();
       "Given an existing issue".
        f(() => MockIssueStore
          .Setup(i => i.FindAsync("1"))
          .Returns(Task.FromResult(fakeIssue)));
       "When it is retrieved".
         f(() =>
         {
           Request.RequestUri = _uriIssue1;
           Response = Client.SendAsync(Request).Result;
           issue = Response.Content.ReadAsAsync<IssueState>().Result;
         });
       "Then a LastModified header is returned".
         f(() =>
         {
           Response.Content.Headers.LastModified
             .ShouldEqual(new DateTimeOffset(new DateTime(2013, 9, 4))); // <1>
         });
       "Then a CacheControl header is returned".
         f(() =>
         {
           Response.Headers.CacheControl.Public
             .ShouldBeTrue(); // <2>
           Response.Headers.CacheControl.MaxAge
             .ShouldEqual(TimeSpan.FromMinutes(5)); // <3>
         });
       "Then a '200 OK' status is returned".
         f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK));
       "Then it is returned".
         f(() => issue.ShouldNotBeNull());
      }
  }
```

示例 8-3 中的测试，与前面所写的获取所有问题的测试略有不同。除了获取单个问题外，这个测试还检验了响应中的 `LastModified` 标头 <1>。`LastModified` 标头稍后将在其他场景用于缓存重验证。这个测试中 `CacheControl` <2> 和 `MaxAge` <3> 标头的预期值也分别为 `Pubilc` 和 5 分钟。

8.4　实现缓存重验证

一旦已缓存的资源表示副本变得陈旧，缓存中间层可以向源服务器发送一个条件 `GET`，重新验证这个副本。条件 `GET` 用到两个响应标头：`If-None-Match` 和 `If-Modified-Since`。`If-None-Match` 对应一个 `Etag` 标头。`Etag` 是一个不透明的值，只有服务器才知道如何重新创建。这个 `Etag` 值可以代表任何东西，但是通常是代表资源版本的一个散列值，可以通过对整个表示的内容进行散列运算，或者只对一部分内容，如时间戳进行运算得到。另一方

面，If-Modified-Since 对应一个 Last-Modified 标头。Last-Modified 是一个时间值，服务器可以用这个时间来判断自上一次提供资源后，这个资源是否发生了变化。

示例 8-4 演示了客户端 / 服务器如何使用前面介绍的这些缓存标头进行请求 / 响应消息交换。

示例 8-4：使用缓存标头的一对请求和响应消息

```
Response ->

Connection close
Date Thu, 02 Oct 2013 14:46:57 GMT
Expires Sat, 01 Nov 2013 14:46:57 GMT
Last-Modified Mon, 29 Sep 2013 15:40:27 GMT
Etag a9331828c518ac6d97f93b3cfdbcc9bc
Content-Type application/json

Request ->

Host localhost
Accept */*
If-Modified-Since Mon, 29 Sep 2013 15:40:27 GMT
If-None-Match a9331828c518ac6d97f93b3cfdbcc9bc
```

缓存中间层可以使用 If-None-Match 和 If-Modified-Since 中的任意一个，判断源服务器上的资源表示是否已发生变化。如果根据这些标头中的数值，资源没有发生变化（If-Modified-Since 对应 Last-Modified，If-None-Match 对应 ETag），服务就会返回一个 HTTP 状态码 304 Not Modified，告诉中间层继续保留已缓存的版本，并刷新过期时间。示例 8-4 展示了两个标头，但是在实际应用中，中间层只使用其中一个。

8.5 为缓存重验证实现条件GET

示例 8-5 展示了我们的第一个测试，这个测试会重新验证已缓存的一个问题资源表示，这个问题在服务器上并未修改。这些测试方法位于 CacheValidation 类中。

```
Scenario: Retrieving an existing issue that has not changed
    Given an existing issue
    When it is retrieved with an IfModifiedSince header
    Then a CacheControl header is returned
    Then a '304 Not Modified' status is returned
    Then it is not returned
```

示例 8-5：验证未改变的缓存副本的单元测试

```
private Uri _uriIssue1 = new Uri("http://localhost/issue/1");

[Scenario]
public void RetrievingNonModifiedIssue()
{
```

```
    IssueState issue = null;

    var fakeIssue = FakeIssues.FirstOrDefault();
    "Given an existing issue".
      f(() => MockIssueStore.Setup(i => i.FindAsync("1"))
          .Returns(Task.FromResult(fakeIssue)));
    "When it is retrieved with an IfModifiedSince header".
      f(() =>
      {
        Request.RequestUri = _uriIssue1;
        Request.Headers.IfModifiedSince = fakeIssue.LastModified; // <1>
        Response = Client.SendAsync(Request).Result;
      });
    "Then a CacheControl header is returned".
      f(() =>
      {
        Response.Headers.CacheControl.Public.ShouldBeTrue();
        Response.Headers.CacheControl.MaxAge.ShouldEqual(TimeSpan.FromMinutes(5));
      });
    "Then a '304 NOT MODIFIED' status is returned".
      f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.NotModified)); // <2>
    "Then it is not returned".
      f(() => Assert.Null(issue));
  }
```

示例 8-5 展示了一个单元测试，在这个测试验证的场景中，源服务器上的资源表示自缓存之后没有发生改变。这个测试模拟了一个缓存中间层的行为，使用预先保存的 `IfModifiedSince` 标头值 <1>，向服务器发送一个条件 GET。在预期验证部分，响应的状态码预期值为 `304 NOT MODIFIED` <2>。

我们需要修改 `IssueController` 类的 `Get` 方法，加入所有的条件 GET 逻辑（参见示例 8-6）。如果服务收到一个带有 `IfModifiedSince` 标头的请求消息，那么应该将标头中的日期值，与受到请求的问题的 `LastModified` 字段进行比较，判断这个问题在最后一次发送给缓存中间层之后，是否发生了改变。

示例 8-6：支持条件 GET 的新版本 Get 方法

```
public async Task<HttpResponseMessage> Get(string id)
{
  var result = await _store.FindAsync(id);
  if (result == null)
    return Request.CreateResponse(HttpStatusCode.NotFound);

  HttpResponseMessage response = null;

  if( Request.Headers.IfModifiedSince.HasValue &&
      Request.Headers.IfModifiedSince == result.LastModified) // <1>
  {
    response = Request
      .CreateResponse(HttpStatusCode.NotModified); // <2>
  }
  else
```

```
  {
    response = Request
      .CreateResponse(HttpStatusCode.OK, _stateFactory.Create(result));
    response.Content.Headers.LastModified = result.LastModified;
  }

  response.Headers.CacheControl = new CacheControlHeaderValue(); // <3>
  response.Headers.CacheControl.Public = true;
  response.Headers.CacheControl.MaxAge = TimeSpan.FromMinutes(5);

  return response;
}
```

示例 8-6 中的新代码检查请求中是否包含 `IfModifiedSince` 标头，以及标头值是否与获取的问题的 `LastModified` 字段值相等 <1>。如果这些条件都满足，那么服务返回一个状态码为 `304 Not Modified` 的响应 <2>。代码的最后更新缓存标头值，并将缓存标头作为响应的一部分返回 <3>。

在我们下一个测试（参见示例 8-7）场景中，源服务器上的资源表示在最后一次由中间层缓存后，发生了改变：

```
Scenario: Retrieving an existing issue that has changed
    Given an existing issue
    When it is retrieved with an IfModifiedSince header
    Then a LastModified header is returned
    Then a CacheControl header is returned
    Then a '200 OK' status is returned
    Then it is returned
```

示例 8-7：验证已改变的缓存副本的单元测试

```
private Uri _uriIssue1 = new Uri("http://localhost/issue/1");

[Scenario]
public void RetrievingModifiedIssue()
{
  IssueState issue = null;

  var fakeIssue = FakeIssues.FirstOrDefault();

  "Given an existing issue".
    f(() => MockIssueStore.Setup(i => i.FindAsync("1"))
        .Returns(Task.FromResult(fakeIssue)));
  "When it is retrieved with an IfModifiedSince header".
    f(() =>
  {
    Request.RequestUri = _uriIssue1;
    Request.Headers.IfModifiedSince = fakeIssue.LastModified
        .Subtract(TimeSpan.FromDays(1)); // <1>
    Response = Client.SendAsync(Request).Result;
    issue = Response.Content.ReadAsAsync<IssueState>().Result;
  });
```

```
    "Then a LastModified header is returned".
      f(() =>
      {
        Response.Content.Headers.LastModified.ShouldEqual(fakeIssue.LastModified);
      });
    "Then a CacheControl header is returned".
      f(() =>
      {
        Response.Headers.CacheControl.Public.ShouldBeTrue();
        Response.Headers.CacheControl.MaxAge.ShouldEqual(TimeSpan.FromMinutes(5));
      });
    "Then a '200 OK' status is returned".
      f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <2>
    "Then it is returned".
      f(() => issue.ShouldNotBeNull()); // <3>
  }
```

和前一个发送条件 GET 的测试相比，这个测试做了一些小改动。这个测试修改了 IfModifiedSince 标头的值，发送的时间不是待验证问题中的 LastModified 字段值，而是一个更旧的时间。在这种情况下，Get 方法的实现会返回一个状态码 200 OK，以及资源表示的一份新副本 <3>。

8.6 冲突检测

我们已经讨论了如何使用条件 GET 重新验证缓存的资源表示，现在要介绍和条件 GET 对应的更新操作：条件 PUT 或条件 PATCH。当同时对一个资源进行多个更新时，我们可以使用条件 PUT/PATCH 检测可能发生的冲突。条件 PUT/PATCH 使用先写 / 先赢（first-write/first-win）的方法解决冲突，也就是说，只有当源服务器上的资源在发送给客户端后没有发生改变时，这个客户端才能对这个资源进行更新操作，否则就会收到一个冲突错误（HTTP 状态码 409 Conflict）。

条件 PUT/PATCH 也使用 If-None-Match 和 If-Modified-SInce 标头代表资源版本，或者与待更新的资源表示形成相关的时间戳。假设两个客户端（X1 和 X2）试图更新同一个资源 R1，下面的步骤详细介绍了冲突检测对这一情况的的处理方法。

(1) 客户端 X1 对 R1（版本 1）执行一个 GET 操作，得到的 HTTP 响应包含了资源表示，标头中包含资源当前版本（此时版本为 V1）的 ETag（也可以使用 Last-Modified 标头）。

(2) 客户端 X2 对同一资源 R1（版本 1）执行一个 GET 操作，得到的资源表示与客户端 X1 相同。

(3) 客户端 X2 对 R1 执行一个 PUT/PATCH 操作，更新资源表示。这个请求包含了资源表示的修改版本，以及带有当前资源版本（V1）的 If-None-Match 标头。更新操作的结果是，服务器返回一个状态码为 OK 的响应，并修改了资源版本号（V2）。

(4) 客户端 X1 对 R1 执行一个 PUT/PATCH 操作。这个请求消息也包含一个带有资源版本 V1

的 If-None-Match 标头。服务器检测到这个资源在以 V1 版本发送之后已经发生了改变，因此返回一个状态码为 409 Conflict 的响应。

8.7 实现冲突检测

示例 8-8 展示了我们的第一个测试，这个测试将在无冲突的情况下更新一个问题，也就是说，IfModifiedSince 值，和问题的 LastModified() 字段中保留部分的值相同。这些测试方法可在 ConflictDetection 类中找到。

```
Scenario: Updating an issue with no conflicts
    Given an existing issue
    When a PATCH request is made with an IfModifiedSince header
    Then a '200 OK' is returned
    Then the issue should be updated
```

示例 8-8：无冲突情况下更新问题的单元测试

```
private Uri _uriIssue1 = new Uri("http://localhost/issue/1");

[Scenario]
public void UpdatingAnIssueWithNoConflict()
{
  var fakeIssue = FakeIssues.FirstOrDefault();

  "Given an existing issue".
    f(() =>
    {
      MockIssueStore.Setup(i => i.FindAsync("1"))
        .Returns(Task.FromResult(fakeIssue));
      MockIssueStore.Setup(i => i.UpdateAsync("1", It.IsAny<Object>()))
        .Returns(Task.FromResult(""));
    });
  "When a PATCH request is made with IfModifiedSince".
    f(() =>
    {
      var issue = new Issue();
      issue.Title = "Updated title";
      issue.Description = "Updated description";
      Request.Method = new HttpMethod("PATCH");
      Request.RequestUri = _uriIssue1;
      Request.Content = new ObjectContent<Issue>(issue,
          new JsonMediaTypeFormatter());
      Request.Headers.IfModifiedSince = fakeIssue.LastModified; // <1>
      Response = Client.SendAsync(Request).Result;
    });
  "Then a '200 OK' status is returned".
    f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK)); // <2>
  "Then the issue should be updated".
    f(() => MockIssueStore.Verify(i => i.UpdateAsync("1",
        It.IsAny<JObject>()))); // <3>
}
```

示例 8-8 展示了第一个测试场景的实现。在这个场景中，IfModifiedSince 标头值设置为待更新问题的 LastModified 属性值 <1>。既然 IfModifiedSince 和 LastModified 值相同，服务器应该检测不到冲突，因此返回状态码 200 OK <2>。最后，问题库中的问题也得到了更新 <3>。

我们需要修改 IssuesController 类中的 Patch 方法，增加示例 8-9 所示的所有条件更新逻辑。

示例 8-9：支持条件更新的新版本 Patch 方法

```
public async Task<HttpResponseMessage> Patch(string id, JObject issueUpdate)
{
    var issue = await _store.FindAsync(id);
    if (issue == null)
        return Request.CreateResponse(HttpStatusCode.NotFound);

    if (!Request.Headers.IfModifiedSince.HasValue) // <1>
        return Request.CreateResponse(HttpStatusCode.BadRequest,
                "Missing IfModifiedSince header");

    if (Request.Headers.IfModifiedSince != issue.LastModified) // <2>
        return Request.CreateResponse(HttpStatusCode.Conflict); // <3>

    await _store.UpdateAsync(id, issueUpdate);
    return Request.CreateResponse(HttpStatusCode.OK);
}
```

示例 8-9 展示了 Patch 方法中的新改动。如果客户端没有发送 IfModifiedSince 标头，那么我们认为这个请求是无效的，直接返回状态码为 400 Bad Request 的响应 <1>；如果客户端发送了 IfModifiedSince 标头,那么请求消息中的 IfModifiedSicne 标头值将与待更新问题的 LastModified 字段比较 <2>，如果二者不同，代码将返回状态码为 409 Conflict 的响应 <3>。如果以上情况皆不是，那么代码最后将更新问题，返回状态码为 200 OK 的响应。

示例 8-10 中展示了下一个测试，测试检测到冲突的场景：

```
Scenario: Updating an issue with conflicts
    Given an existing issue
    When a PATCH request is made with an IfModifiedSince header
    Then a '409 CONFLICT' is returned
    Then the issue is not updated
```

示例 8-10：有冲突情况下更新问题的单元测试

```
[Scenario]
public void UpdatingAnIssueWithConflicts()
{
  var fakeIssue = FakeIssues.FirstOrDefault();

  "Given an existing issue".
    f(() =>
    {
      MockIssueStore.Setup(i => i.FindAsync("1"))
```

```
            .Returns(Task.FromResult(fakeIssue));
        });
      "When a PATCH request is made with IfModifiedSince".
        f(() =>
        {
          var issue = new Issue();
          issue.Title = "Updated title";
          issue.Description = "Updated description";
          Request.Method = new HttpMethod("PATCH");
          Request.RequestUri = _uriIssue1;
          Request.Content = new ObjectContent<Issue>(issue,
            new JsonMediaTypeFormatter());
          Request.Headers.IfModifiedSince = fakeIssue.LastModified.AddDays(1); // <1>
          Response = Client.SendAsync(Request).Result;
        });
      "Then a '409 CONFLICT' status is returned".
        f(() => Response.StatusCode
          .ShouldEqual(HttpStatusCode.Conflict)); // <2>
      "Then the issue should be not updated".
        f(() => MockIssueStore.Verify(i =>
          i.UpdateAsync("1", It.IsAny<JObject>(), Times.Never())); // <3>
    }
```

示例 8-10 中的代码，测试了检测到冲突的场景。我们将 `IfModifiedSince` 标头的值设置为待更新问题的 `LastModified` 属性值加一天 <1>。由于 `IfModifiedSince` 和 `LastModified` 值不同，测试预期服务器返回一个状态码为 `409 Conflict` 的响应 <2>。最后，测试还验证了问题库中的问题并没有得到更新 <3>。

8.8 变更审计

我们的 Web API 要支持的另一个功能是，识别创建新问题或者更新已有问题的用户或客户端，这意味着 API 实现要使用预先定义的认证方案，基于应用程序键、用户名 / 密码、HMAC（Hash-based Message Authentication Code，基于散列的消息认证码）或安全令牌（如 OAuth），对客户端进行认证。

使用应用程序键可能是最简单的实现方法。每个客户端应用程序都由一个简单固定的应用程序键进行标识。这种认证机制可能比较弱，但是我们的服务提供的并非敏感数据，任何有应用程序键的人都可以使用这些数据。公共服务，如谷歌地图或公共图片搜索（例如 Instagram 中的搜索），大多使用应用程序键认证方案。使用应用程序键的唯一目的，是识别客户端，以便对其应用不同的服务水平协议（service-level agreement），例如 API 配额或可用性。得到了客户端的应用程序键，任何人都可以对其进行模拟。

HMAC 与应用程序键认证的机制相似，但是使用基于密钥的加密算法，以避免应用程序键认证方案中的模拟问题。与基本认证不同的是，密钥或密码不会在每个消息中以明文发送。HMAC 或散列是基于密钥，使用 HTTP 请求消息的部分内容生成的，并包含在认证标头中。服务器可以通过认证标头中附带的 HMAC，验证客户端的身份。这种认证模

式特别适合云计算。在云计算环境中，服务提供方，如 AWS（Amazon Web Services）或 Windows Azure，使用一个键值来识别用户，以提供正确的服务和私有数据。不管用户使用什么客户端应用程序来使用服务和数据，这个键值的主要目的是识别用户的身份。虽然存在好几种 HMAC 认证的实现，我们在这里将只介绍一个对 HMAC 认证进行标准化的新兴规范——Hawk。

最后一种认证方案基于安全令牌，可能是最为复杂的一种方案。OAuth 就是这种方案的一种，设计用于在 Web 2.0 中进行授权认证。拥有数据的服务可以使用 OAuth 与其他服务或应用程序共享数据，而不会泄露数据所有者的身份。

第 15 章将详细讨论以上提到的这些认证方案。在本章中，我们将使用 Hawk，在设置问题的审计信息前，对客户端应用程序进行认证。

8.9 使用Hawk认证实现变更审计

我们的第一个测试将创建一个新问题，这个问题带有关于创建者的审计信息。因此，这个测试也将实现使用 Hawk，对客户端进行 HMAC 认证。这些测试的代码位于 `CodeAuditing` 类中。

```
Scenario: Creating a new issue
    Given a new issue
    When a POST request is made with an Authorization header containing the user
        identifier
    Then a '201 Created' status is returned
    Then the issue should be added with auditing information
    Then the response location header will be set to the resource location
```

为了在实现中增加 Hawk 认证，我们要使用 GitHub（https://github.com/pcibraro/hawknet）上一个现成的开源实现 HawkNet。HawkNet 支持与多个 Web API 框架的集成，其中就包括 ASP.NET Web API。HawkNet 通过 HTTP 消息处理程序，实现了与 ASP.NET WEb API 的集成（参见示例 8-11）。客户端使用一个处理程序，在每个发出的调用中自动加入 Hawk 认证标头，服务器端使用另一个处理程序，验证标头，对客户端进行认证。

示例 8-11：在 `HttpClient` 实例中注入 `HawkClientMessageHandler`

```
Credentials = new HawkCredential
{
    Id = "TestClient",
    Algorithm = "hmacsha256",
    Key = "werxhqb98rpaxn39848xrunpaw3489ruxnpa98w4rxn",
    User = "test"
}; // <1>

var server = new HttpServer(GetConfiguration());
Client = new HttpClient(new HawkClientMessageHandler(server, Credentials)); // <2>
```

示例 8-11 展示了如何在测试使用的 `HttpClient` 实例中注入 `HawkClientMessageHandler`。HawkNet 使用 `HawkCredential` 类进行各种配置，设定如何生成 Hawk 标头。我们的测试配置这个 `HawkCredential` 类使用 SHA-256 算法生成 HMAC，还配置了密钥、应用程序 `id`（`TestClient`），以及与这个密钥关联的用户（`test`）<1>。`HawkCredential` 类进行实例化和配置之后，就传递给注入 `HttpClient` 实例的 `HawkClientMessageHandler` <2>。

除此之外，服务器也需要配置对应的消息处理程序，用于验证标头和认证客户端。HawkNet 为此提供一个 `HawkMessageHandler` 类，这个类可以作为路由配置的一部分注入，或者作为全局处理程序注入（参见示例 8-12）。

示例 8-12：在路由配置中注入 `HawkMessageHandler`

```
Credentials = new HawkCredential
{
    Id = "TestClient",
    Algorithm = "hmacsha256",
    Key = "werxhqb98rpaxn39848xrunpaw3489ruxnpa98w4rxn",
    User = "test"
};

var config = new HttpConfiguration();

var serverHandler = new HawkMessageHandler(new HttpControllerDispatcher(config),
(id) => Credentials);

config.Routes.MapHttpRoute("DefaultApi", "{controller}/{id}", new { id =
RouteParameter.Optional }, null, serverHandler);
```

一旦发送和认证 Hawk 标头的处理程序就位，我们就可以开始编写测试，测试创建问题的第一个场景了。示例 8-13 展示了这个测试的最终实现。

示例 8-13：创建一个新问题的实现

```
[Scenario]
public void CreatingANewIssue()
{
  Issue issue = null;

  "Given a new issue".
    f(() =>
      {
        issue = new Issue
        {
          Description = "A new issue",
          Title = "A new issue"
        };

        var newIssue = new Issue { Id = "1" };

        MockIssueStore
          .Setup(i => i.CreateAsync(issue, "test"))
```

```
            .Returns(Task.FromResult(newIssue));
        });
    "When a POST request is made with an Authorization header containing the user
    identifier".
      f(() =>
      {
          Request.Method = HttpMethod.Post;
          Request.RequestUri = _issues;
          Request.Content = new ObjectContent<Issue>(issue,
            new JsonMediaTypeFormatter());
          Response = Client.SendAsync(Request).Result;
      });
    "Then a '201 Created' status is returned".
      f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.Created));
    "Then the issue should be added with auditing information".
      f(() => MockIssueStore.Verify(i => i.CreateAsync(issue, "test"))); // <1>
    "Then the response location header will be set to the resource location".
      f(() => Response.Headers.Location.AbsoluteUri.ShouldEqual
          ("http://localhost/issue/1"));
}
```

这个测试主要验证的是，创建的新问题在问题库中存储正确，并包含了认证用户信息 test<1>。我们修改了 IIssueStore 接口的 CreateAsync 方法，增加了一个参数，代表创建问题的用户。IssueController 类的 Post 方法负责从认证用户得到这个参数值，并传递给 CreateAsync（参见示例 8-14）。

示例 8-14：更新版本的 Post 方法

```
[Authorize]
public async Task<HttpResponseMessage> Post(Issue issue)
{
    var newIssue = await _store.CreateAsync(issue, User.Identity.Name); // <1>
    var response = Request.CreateResponse(HttpStatusCode.Created);
    response.Headers.Location = _linkFactory.Self(newIssue.Id).Href;
    return response;
}
```

认证用户可以通过 User.Identity 属性得到。HawkMessageHanlder 在验证了收到的 Authorization 标头后，设置 User.Identity 属性的值。这个用户和接收到的问题一起作为参数传递给 CreateAsync 方法 <1>。此外，Post 方法受到 Authorize 属性修饰，以拒绝任何匿名的调用。

```
Scenario: Updating an issue
    Given an existing issue
    When a PATCH request is made with an Authorization header containing the
    user identifier
    Then a '200 OK' is returned
    Then the issue should be updated with auditing information
```

验证这个场景的测试实现，也需要检查 IIssueStore 中的问题修改是否正确，以及是否带有认证用户信息（参见示例 8-15）。

示例 8-15：更新问题的测试实现

```
[Scenario]
public void UpdatingAnIssue()
{
  var fakeIssue = FakeIssues.FirstOrDefault();

  "Given an existing issue".
    f(() =>
    {
      MockIssueStore
        .Setup(i => i.FindAsync("1"))
        .Returns(Task.FromResult(fakeIssue));
      MockIssueStore
        .Setup(i => i.UpdateAsync("1", It.IsAny<Object>(), It.IsAny<string>()))
        .Returns(Task.FromResult(""));
    });
  "When a PATCH request is made with an Authorization header containing the user
  identifier".
    f(() =>
    {
      var issue = new Issue();
      issue.Title = "Updated title";
      issue.Description = "Updated description";
      Request.Method = new HttpMethod("PATCH");
      Request.Headers.IfModifiedSince = fakeIssue.LastModified;
      Request.RequestUri = _uriIssue1;
      Request.Content = new ObjectContent<Issue>(issue, new JsonMediaTypeFormatter());
      Response = Client.SendAsync(Request).Result;
    });
  "Then a '200 OK' status is returned".
    f(() => Response.StatusCode.ShouldEqual(HttpStatusCode.OK));
  "Then the issue should be updated with auditing information".
    f(() => MockIssueStore.Verify(i => i.UpdateAsync("1", It.IsAny<JObject>(),
        "test"))); // <1>
}
```

`IIssueStore` 接口的 `UpdateAsync` 方法也改为多使用一个参数，代表更新问题的用户 <1>。

示例 8-16 展示了 `Patch` 方法的修改版本，方法中对 `IIssueStore` 的 `UpdateAsync` 调用进行了修改，传入认证用户参数。

示例 8-16：更新版本的 `Patch` 方法

```
[Authorize]
public async Task<HttpResponseMessage> Patch(string id, JObject issueUpdate)
{
  var issue = await _store.FindAsync(id);
  if (issue == null)
      return Request.CreateResponse(HttpStatusCode.NotFound);

  if (!Request.Headers.IfModifiedSince.HasValue)
      return Request.CreateResponse(HttpStatusCode.BadRequest,
           "Missing IfModifiedSince header");
```

```
    if (Request.Headers.IfModifiedSince != issue.LastModified)
        return Request.CreateResponse(HttpStatusCode.Conflict);

    await _store.UpdateAsync(id, issueUpdate, User.Identity.Name); // <1>
    return Request.CreateResponse(HttpStatusCode.OK);
}
```

8.10 跟踪

在一个不具备集成开发环境或者代码调试工具不可用的环境里，或者在开发早期 API 性能还不稳定，有一些随机的难以确定的问题发生时，要解决问题或者调试 Web API，跟踪是一项不可替代的功能。ASP.NET Web API 自带一个跟踪架构，可以用于跟踪框架自身的任何行为，或者 Web API 实现中的任何定制代码。

ASP.NET Web API 跟踪架构的核心组件或服务是 `System.Web.Http.Tracing.ITraceWriter` 接口，这个接口只包含一个方法 `Trace`，用于生成一个新的跟踪条目。

示例 8-17：`ITraceWriter` 接口定义

```
public interface ITraceWriter
{
  void Trace(HttpRequestMessage request, string category, TraceLevel level,
  Action<TraceRecord> traceAction);
}
```

`Trace` 方法使用如下参数。

- `request`

  与跟踪相关的请求消息。

- `category`

  与跟踪条目相关的类别，便于组织或过滤跟踪条目。

- `level`

  与条目相关的详细级别，可以用于过滤跟踪条目。

- `traceAction`

  对生成跟踪条目的方法的委托。

虽然这个跟踪架构没有绑定到 .NET 中的任何日志框架——如 Log4Net、NLog 或 Enterprise Library Logging——但是这个架构提供了一个默认的日志实现，称为 `System.Web.Http.Tracing.SystemDiagnosticsTraceWritter`。它使用 `System.Diagnostics.Trace.TraceSource`。要使用其他的日志框架，代码必须提供 `ITraceWritter` 服务接口的实现。

示例 8-18 展示了如何在 Web API 配置对象中注入一个定制实现。

示例 8-18：ITraceWritter 配置

```
HttpConfiguration config = new HttpConfiguration(); config.Services.
Replace(typeof(ITraceWriter), new SystemDiagnosticsTraceWriter());
```

8.11　实现跟踪

我们有一个场景，测试 IssueController 类中所有方法的大致跟踪。测试代码位于 Tracing 类中。

```
Scenario: Creating, Updating, Deleting, or Retrieving an issue
    Given an existing or new issue
    When a request is made
    When the diagnostics tracing is enabled
    Then the diagnostics tracing information is generated
```

在对这个场景编写测试前，我们要做的第一件事情是配置 ITraceWriter 的一个实例，用于检验跟踪功能是否真的起作用。详见示例 8-19。

示例 8-19：测试的 ITraceWriter 配置

```
public abstract class IssuesFeature
{
  public Mock<ITraceWriter> MockTracer;

  public IssuesFeature()
  {
  }

  private HttpConfiguration GetConfiguration()
  {
    var config = new HttpConfiguration();

    MockTracer = new Mock<ITraceWriter>(MockBehavior.Loose);

    config.Services.Replace(typeof(ITraceWriter), MockTracer.Object); // <1>

    return config;
  }
}
```

示例 8-19 展示了如何在 Web API 使用的 HttpConfiguration 实例中，注入一个模拟实例 <1>。我们的测试将使用这个模拟实例（参见示例 8-20），验证控制器方法对 Trace 方法的调用。

示例 8-20：跟踪测试实现

```
public class Tracing : IssuesFeature
{
  private Uri _uriIssue1 = new Uri("http://localhost/issue/1");
```

```
    [Scenario]
    public void RetrievingAnIssue()
    {
      IssueState issue = null;
      var fakeIssue = FakeIssues.FirstOrDefault();

      "Given an existing or new issue".
        f(() =>
        {
          MockIssueStore
            .Setup(i => i.FindAsync("1"))
            .Returns(Task.FromResult(fakeIssue));
        }
      "When a request is made".
        f(() =>
        {
          Request.RequestUri = _uriIssue1;

          Response = Client
            .SendAsync(Request)
            .Result;

          issue = Response.Content
            .ReadAsAsync<IssueState>()
            .Result;
        });
      "When the diagnostics tracing is enabled".
        f(() =>
        {
          Configuration.Services
            .GetService(typeof(ITraceWriter)).ShouldNotBeNull(); // <1>
        });
      "Then the diagnostics tracing information is generated".
        f(() =>
        {
          MockTracer.Verify(m => m.Trace(It.IsAny<HttpRequestMessage>(), // <2>
            typeof(IssueController).FullName,
            TraceLevel.Debug,
            It.IsAny<Action<TraceRecord>>()));
        });
    }
  }
```

示例 8-20 中的测试代码验证了当前 `HttpConfiguration` 实例中配置了 `ITraceWriter` 服务，并检验 `IssueController` 类（参见示例 8-21），向配置的模拟实例发送了跟踪消息。

示例 8-21：`IssueController` 中的跟踪语句

```
public async Task<HttpResponseMessage> Get(string id)
{
    var tracer = this.Configuration.Services.GetTraceWriter(); // <1>

    var result = await _store.FindAsync(id);
    if (result == null)
```

```
        {
            tracer.Trace(Request,
                TraceCategory, TraceLevel.Debug,
                    "Issue with id {0} not found", id); // <2>

            return Request.CreateResponse(HttpStatusCode.NotFound);
        }

        .....
    }
```

`HttpConfiguration` 类提供了一个扩展方法或捷径，可以获得配置的 `ITraceWriter` 实例，因此可以在实现中由定制代码使用。示例 8-21 展示了如何修改 `IssueController` 类，以获得 `ITraceWriter` 的引用 <1>，然后用来在返回响应之前输出跟踪信息，说明问题未找到 <2>。

8.12　小结

本章介绍了对现有 Web API 进行改进的几个重要方面，如缓存、冲突管理、审计和跟踪。虽然有些功能在某些情况下并不适用，但是了解了这些功能带来的好处，你就可以正确使用它们。

第 9 章

构建客户端

一个巴掌拍不响。

我们已经说过，贯穿这本书的目标是构建一个可演化的系统。到目前为止，我们的大部分精力都放在构建 API 上，关注如何使客户端能够保持松耦合，以实现可演化系统的目标。遗憾的是，服务器 API 只能为松耦合提供条件，并不能防止紧耦合的发生。不管我们如何使用超媒体、标准媒体类型和自描述性消息，都不能阻止客户端使用硬编码的 URI，也不能保证客户端不假定自己了解响应消息的内容类型和语义。

公布一个 Web API，一定程度上是为了给 API 使用者提供指导意见，说明如何使用服务能够更好地利用 Web 架构的功能。我们需要告诉客户端的开发者，如何使用 API，避免使用那些可能发生变化的依赖物。

为客户端开发者提供指导，我们可以编写文档，但是文档通常只能解决部分问题。API 提供者希望 API 使用者的体验尽可能地愉快，因此经常会提供客户端程序库，企图使客户端开发者能够很快开始工作。可悲的是，人们常常认为，如果不能在五分钟内学会使用一个软件程序库，那么这个程序库肯定写得不好。可惜，这种对易用性的优选法没有认识到简单和容易之间的细微差别（http://www.infoq.com/presentations/Simple-Made-Easy）。

在努力提高 API 易用性的过程中，API 提供者经常会创建一些客户端程序库，对基于 HTTP 的 API 进行封装，从而失去了 Web 架构的许多优点，最后导致客户端应用程序与客户端程序库紧密耦合，而客户端程序库又与服务器 API 紧密耦合。在本章中，我们将更详细地讨论这种客户端程序库的负面影响，然后介绍一种替代方法，既和封装 API 一样易于使用，又不会导致同样的问题。接着，我们将讨论构建客户端逻辑和管理客户端状态的技

术，使客户端能够灵活适应变化。

9.1 客户端程序库

使用客户端程序库，目的是提高抽象度，使客户端应用程序可以编写与应用程序域相关的代码。通过重用标准代码，创建 HTTP 请求和解析响应，开发者可以将精力集中在功能实现上。

9.1.1 封装库

API 封装库的例子比比皆是。如果你搜索 Programmable Web（http://www.programmableweb.com/）这样的网站，会得到很多的结果，指向的网站都提供几十种不同编程语言的客户端程序库。有的时候，这些客户端程序库来自 API 提供者，但是在如今，很多的程序库是社区参与者提供的。API 提供者发现，维护所有这些不同版本的程序库，需要的工作量太大。

无论一个客户端 API 封装程序的具体提供者是谁，代码通常是这样的：

```
var api = new IssueTrackingApi(apiToken);
var issue = api.GetIssue(issueId);
issue.Description = "Here is a description for my issue";
issue.Save();
```

这段代码有一个最基本的问题：客户端开发者不再了解，这四行代码中的哪一行会发出网络请求，哪一行不发出请求。把发出网络请求的标准代码抽象出去，这种做法并没有问题，但是如果完全把这些高延迟交互发生的位置隐藏起来，客户端应用程序的开发者就很难写出网络效率高的应用程序。

1. 可靠性

开发基于网络的应用程序，面对的挑战之一是网络不可靠，在无线网络和蜂窝移动通信环境中这个问题尤为严重。作为一个应用程序协议，HTTP 具有一些功能，可以使应用程序容忍这样不可靠的传输介质。调用如 `GetIssue` 或 `Save` 这样的方法，可能得到的结果有两种：一种是成功以及预期的返回类型，另一种是异常。HTTP 将请求区分为安全和不安全的，幂等和非幂等的。根据这些分类，客户端应用程序可以解释标准的状态码，决定在失败时采取何种补救措施。在面向对象和过程化的编程中，并没有这样的约定。我可以猜测 `GetIssue` 是安全的，`Save` 是不安全而且（在这个代码示例中）是幂等的。但是，我可能只是根据方法名，猜想其底层的客户端行为，而做出了这样的推断。将 HTTP 请求隐藏在过程调用之后，就隐藏了 HTTP 的可靠性功能，迫使客户端开发者重新创建自己的约定和标准，重新获得这种可靠性。

客户端程序库可以使用某些可靠性机制，如重试，一些设计比较好的程序库可能会采取这

种做法。但是，编写程序库的开发人员并不知道客户端应用程序的可靠性需求。同一种矫正行为，对一个客户端应用可能是有益的，而对另一个应用并不适用。在夜间抓取数据源的批处理程序可能很乐于将一个请求重试两三次，每个请求间隔等待几分钟，但如果等待响应的是人类，恐怕就不会这么耐心等待响应结果了。

HTTP 请求的结果并不仅仅会影响处理服务临时中断的方式。请求结果可能有：请求的资源不存在（410 Gone）、资源已移走（303 See other）、禁止访问（403 Forbidden）或者需要等待资源创建（202）。响应的内容类型（Content-Type）也可能和上次请求时不同。所有这些都不是执行请求的失败结果。对于一个预期会随着时间演化的系统，这些都是在对其请求信息时得到的有效响应。如果我们要构建一个能够多年使用的可靠客户端，就需要处理所有这些情况。

我经常看到，API 文档对一个特定 API 资源可能返回的状态码进行描述。不幸的是，API 返回的状态码是不受约束的。在每个 HTTP 请求的处理中，都有许多中间层的参与：客户端连接程序、代理、缓存、负载均衡、反向代理和应用程序中间件，诸如此类。任何中间层都可能中断一个请求，返回另外一些状态码。客户端需要能够处理对所有资源的所有响应码。在构建客户端时，如果假定一个资源从来不会返回状态码 303，那么这种做法和硬编码 URI 一样，都引入了隐藏的耦合。

2. 响应类型

像 GetIssue 这样的方法，假定返回的响应可以转换为一个 Issue 对象。通过在客户端程序库中加入一些基本的内容协商代码，你可以实现一定的灵活性，以处理 JSON 最终由其他一些传输格式替代（正如 XML 如今已经不再流行）的情况。但是，有些可以利用 HTTP 实现的功能，在这种方法严格受限的合约中是无法实现的。请看下面的请求：

```
GET /IssueSearch?priority=high&AssignedTo=Dave
```

在过程化的封装程序库中，对应的方法签名会是这样的：

```
public List<Issue> SearchIssues(int priority, string assignedTo);
```

返回资源的媒体类型可能是 Collection+Json，这种格式非常适合用于返回列表。但是，如果服务器发现满足过滤条件的只有一个问题呢？服务器可能会决定不返回只有一个 Issue 的 Collection+Json 列表，而是返回一个 Issue 完整的 application/Issue+Json 表示。过程化的程序库无法支持这种灵活性，除非是使用 object 类型的返回值，而这又不符合强类型封装程序库的目标。API 开发者可能出于各种原因，不希望在响应中引入这种变动，但是在某些情况下这种功能是极有价值的。例如：带有一列条目和相关支出凭据的开支报告。这些凭据可能是 PDF 文件、TIFF 文件、位图、HTML 页面或电子邮件。你不可能在一个方法签名中支持所有这些资源的强类型表示。

3. 生存期

在之前的客户端代码片段中，我们初始化了一个 Issue 对象。这个对象的状态来自服务器返回的资源表示。服务器返回的资源表示很可能带有一个缓存控制标头。如果缓存控制标头如下段所示，那么服务器声明了数据在随后的 60 秒内是有效的：

```
Cache-Control: private;max-age=60
```

如果存在适当的缓存支持，那么在随后 60 秒内，对这个 Issue 的任何获取操作都会不会产生真正的网络往返通信，而是从缓存返回资源表示。60 秒之后，对这个 Issue 的任何获取操作都将产生一个网络请求，得到最新的信息。如果把资源表示数据复制到一个本地对象中，忽略返回的表示，就把这个数据的生存期与对象的生存期绑定在了一起，60 秒的 max-age 信息失效。不管信息如何陈旧，资源表示数据会一直得到保存和重用，直到这个对象销毁，新的对象生成。在处理并发问题时，由服务器控制数据的生存期是很重要的。服务器是数据的所有者，对数据的易变性有最准确的理解。如果依赖客户端制定缓存规则，客户端就需要对服务器上的数据有比通常所需更多的了解。

如果客户端程序库建立对象关系，那么包含陈旧数据的对象实例会带来更严重的问题。在封装 API 中，你经常看到用一个对象获取资源表示，以填充相关的其他对象，常见做法是使用基于属性的延迟加载：

```
var issue = api.GetIssue(issueId);
var reportedByuser = issue.ReportedBy;
```

在获取数据库信息的 ORM（Object-Related Mapping，对象关系映射）程序库中，这种加载属性的方法极为常见。但是，在 API 封装库中这样做，会将 User 对象的生存期与 Issue 对象的生存期绑定。也就是说，你不仅忽略了 HTTP 的缓存生存期信息，还在数据表示之间创建了生存期依赖，其结果与我们希望的正好相反。一个问题资源很可能比用户资源修改更为频繁，用户资源可能由多个问题资源重用。可是，我们的对象关系把 User 的生存期和 Issue 的生存期绑定，这不是最优的方式。为了解决这个问题，ORM 库经常会创建新的生存期范围，称为*会话*（session）或*工作单元*（unit of work），由这些容器来管理对象生存期。这种解决方法给客户端增加了不必要的复杂度。我们原本可以利用 HTTP 协议，使用服务器提供的缓存信息来避免这些问题，为此我们不能再将 HTTP 隐藏在封装程序后面。

4. 每个人都有自己的风格

客户端封装程序库的另一个问题是，这些库各自维护自身的协议状态，使 API 的使用更加混乱。通过与客户端程序库的一系列交互，未来交互基于存储的状态进行修改。这些存储的状态可能是认证信息，或者其他修改请求的偏好设置。不同的 API 各自实现自己的客户端程序库，创建自己的交互模型，增加了开发者的学习曲线。当客户端程序库必须使用多个 API 才能完成工作时，这种风格差异更加令人烦恼。访问的远程接口遵循标准统一的 HTTP 接口，却需要使用多个行为各异的客户端程序库，简直能令人抓狂。

糟糕的是，很多这种程序库是全有或全无的选择。你要么使用原始的 HTTP，要么全都使用 API。通常，你无法访问封装库发送的 HTTP 请求，进行任何细微的修改。响应也是一样：如果客户端程序库不提供响应中的某些部分，你就没有办法获得这些信息。这通常意味着，如果服务器端的 API 进行了细微的修改，那么你必须改用客户端程序库的新版本，或者至少在升级之前无法使用 API 的新功能。

5. 不适合超媒体

超媒体驱动的 API 提供的资源是动态的。为这种 API 创建 API 封装程序难度特别大。你没有办法定义一个类，有时包含方法，有时又不包含。我相信，超媒体驱动 API 之所以特别少见，原因之一就是大多数人找不到在客户端使用超媒体 API 的简便方法。

超媒体 API 的真正不同之处在于，API 提供的分布式功能是以一段数据的形式传递给客户端的。这些功能表现为返回的资源表示中的嵌入连接。把函数当作数据进行操作并不是一个新的概念，但是，在超媒体中，我们把链接看作代表远程调用的一段数据。

在下一节中，我们要讨论构建客户端程序库的另一种方式，将链接提升为第一类概念，可以用于对 API 进行远程调用。这种方式更适合用于超媒体驱动的 API，但是对于那些没有遵循 REST 全部约束的 API，使用这种方式也可以非常高效地进行访问。

9.1.2　链接用作函数

一个链接中最重要的部分是 URL。但是，URL 本身只是一个标识符，需要由服务器解释。这个标识符的重要性由*链接关系类型*（link relation type）决定。如果不理解一个链接的用途，客户端应用程序就很难使用这个链接。

让我们再来看 `stylesheet` 链接关系。`stylesheet` 的链接关系规范只是简单地说这种链接“指向一个样式表”。但是，Web 浏览器具有明确的逻辑，会自动使用 `GET` 方法对这种类型的链接进行解引用，使用返回的资源表示对内容文档进行显示上的修改。客户端可以对特定的链接关系类型选择使用任何逻辑。

绝大多数的链接关系类型都不指定客户端应该如何处理一个响应。但是，有些关系对于如何激活链接，声明了某种支持的协议。例如：`search`、`oauth2-token` 和 `oauth2-authorize`。

将链接实现为一个类，我们可以在其中包含创建一个 HTTP 请求，以及在某些情况下处理返回响应所需的操作。

```
    var tokenLink = new OAuth2TokenLink
{
    Target = new Uri("https://login.live.com/oauth20_token.srf"),
    RedirectUri = new Uri("https://login.live.com/oauth20_desktop.srf"),
    ClientId = "000000007C0B306F",
    ClientSecret = "LsKOqUbIv5HSPHt2OM9Z4Ay219Mf-DNA",
```

```
        GrantType = "authorization_code",
        AuthorizationCode = "3eab54b1-86fa-4596-ce5e-91cb4e55bbd3"
    };

    var client = new HttpClient();
    var response = await client.SendAsync(tokenLink.CreateRequest());
    var body = await response.Content.ReadAsStringAsync();

    if (response.IsSuccessStatusCode)
    {
        var token = OAuth2TokenLink.ParseTokenBody(body);
    }
    else
    {
        var error = OAuth2TokenLink.ParseErrorBody(body);
    }
```

在这个例子中，OAuth2TokenLink 代表指向一个 OAuth 令牌生成资源的链接。这个链接类提供了发起请求所需的所有参数。这些参数可以定义为 .NET 中任意适合的类型。将这些信息转换成 HTTP 请求所需格式，其中的实现细节都被抽象了出去。

通过调用 CreateRequest，OAuth2TokenLink 类创建了一个 HttpRequestMessage 实例，其方法为 POST，Content 属性为填充了所需全部参数的 FormEncodedUrlContent 实例。随后这个 HttpRequestMessage 实例可以像任何其他请求一样使用。我们可以重用带有通常 DefaultRequestHeaders 和 MessageHandlers 的标准 HttpClient，发送这个请求。

一旦我们获得了 HttpResponseMessage，就可以使用 OAuth2TokenLink 解析响应正文。采用这种方式，我们将所有的链接语义都封装在了链接类中，而且程序库中的类也不必处理发送请求的机制。

1. 服务反模式

客户端封装库的开发者通常会定义一个服务类，提供一组对应于远程资源的方法。例如：如果我们要为 Issue API 创建一个封装库，代码可能如下所示：

```
public class IssueApi {

       public Issue GetIssue(int id) {...}
       public Issue CreateIssue(IssueDto issueInfo) {...}
       public List<Issue> GetOpenIssues() {...}
       public List<Issue> GetMyIssues
       (int userId) {...}

}

var issueApi = new IssueApi("http://example.org/issueApi");
var issue = issueApi.GetIssue(77);

List<Issues> openIssues = issueApi.GetOpenIssues();
```

这种实现方式有一个问题：整个 API 是由服务 API 类决定的。可用的资源、返回类型和参

数都定义在客户端程序库中。

我们可以使用 IssueLink 和 IssuesLink 类实现同样的功能。要访问这些资源，还有更为通用的方法，但是为了简化讨论，我们只考虑 IssueLink 和 IssuesLink。

为了访问问题和问题列表，我们可以使用如下代码：

```
var httpClient = new HttpClient();

var issueLink = new IssueLink() {
        Target = new Uri("http://example.org/issueApi/Issue/{id}"),
        Id = 77
}

var issue = issueLink.ParseResponse(
            await httpClient.SendAsync(issueLink.CreateRequest()));

var issuesLink = new IssuesLink() {
        Target = new Uri("http://example.org/issueApi/OpenIssues")
}

List<Issues> issues = issuesLink.ParseResponse(
            await httpClient.SendAsync(issueLink.CreateRequest()));
```

通过确保我们的 API 耦合只限于链接类型，而非具体的资源，当 API 添加更多资源（如 /ClosedIssues、/CriticalIssues 和 /LateIssues）时，我们相信 IssueLink 还能够正常工作。

```
var httpClient = new HttpClient();

var closedIssuesLink = new IssuesLink {
        Target = new Uri("http://acme.org/issueApi/ClosedIssues"),
};

List<Issues> closedIssues = issuesLink.ParseResponse(
            httpClient.SendAsync(closedIssuesLink.CreateRequest()));

var criticalIssuesLink = new IssuesLink {
        Target = new Uri("http://acme.org/issueApi/CriticalIssues"),
};

List<Issues> criticalIssues = issuesLink.ParseResponse(
            httpClient.SendAsync(criticalIssuesLink.CreateRequest()));

var lateIssuesLink = new IssuesLink {
        Target = new Uri("http://acme.org/issueApi/LateIssues"),
};

List<Issues> lateIssues = issuesLink.ParseResponse(
            httpClient.SendAsync(lateIssuesLink.CreateRequest()));
```

上面这些请求的语义都是相同的，都是一个 GET 请求，返回包含一列问题的媒体类型。虽

然每个链接返回不同的问题子集，但是这并不影响链接的语义。在封装 API 中，如果要实现同样的操作，就必须在客户端程序库中创建新的方法，以提供这些资源。API 的使用者必须等待客户端程序库的新版本发布，并且要更新客户端代码，然后才能访问这些新资源。

为了规避难以向客户端提供新资源的问题，人们经常试图在 API 中提供复杂的查询功能。这种做法不仅会造成对查询语法的耦合，而且还会带来另外一个问题，这寥寥几个查询参数可能打开大量的潜在资源，而这些资源中的一些生成代价很高。很多 API 提供者已经开始意识到，向第三方提供任意查询的功能很难平衡得失。一种更容易控制的方式是，提供少量高度优化的资源，支持大部分的功能。但是，在 API 中添加新资源以满足新需求，这种能力依然是至关重要的。

2. 反序列化链接

一旦你接受了在资源表示中嵌入链接的概念，还可以实现资源表示的反序列化，自动生成链接实例，你要做的只是从资源表示的对象模型中获取这些链接。

当使用的链接关系没有专门的媒体类型时，资源表示的反序列化代码需要使用某种形式的链接工厂，以创建正确类型的链接。你可以用链接关系值查询一个类型字典表，确定需要实例化的正确类型。

3. 分离请求和响应

调用封装类方法和使用链接发起请求，二者最大的区别之一是请求和响应的分离。使用链接发起请求分为清晰的两步：首先，创建请求，发送给源服务器；随后，可选的，将响应传递给链接进行处理。将请求和响应分开具有若干好处。发起 HTTP 请求相对来说是非常重大的操作，因此，所有使用 `HttpClient` 的 HTTP 请求都是异步的。异步操作的请求和响应代码是分开的。最近版本的 C# 和 .NET 可以从语法上隐藏这种分隔，但是在根本上，这种分离依然是存在的。将链接的请求和响应处理分开，更加符合这种异步操作的特点。

使用封装 API 时，我们认为 HTTP 请求将在封装类方法中进行完全的处理。这意味着，访问一个资源的每个方法都必须处理 API 返回的状态码中所有不是 `2XX` 的响应。封装程序库必须决定如何处理 `3XX` 重定向、`401 Unauthroized` 响应和 `503 Server Unavailable` 响应。在使用多个 API 时，因为不同的封装 API 处理这些响应的方式可能会不同，会使事情变得更为复杂。

使用链接类，你可以先检查不是 `2XX` 的状态，然后再将响应传递给链接进行处理。这样可以比较容易对重定向和错误状态进行一致的、集中的处理。

```
var httpClient = new HttpClient();

var issueLink = new IssueLink() {
        Target = new Uri("http://example.org/issueApi/Issue/{id}"),
        Id = 77
}
```

```
var response = await httpClient.SendAsync(issueLink.CreateRequest());

if (response.IsSuccessStatusCode) {
    var issue = issueLink.ParseResponse(response);
} else {
        GlobalNonSuccessResponseHandler.Handle(response)
}
```

HttpClient 类具有可扩展的处理程序管道，可以更容易、更清晰地添加这种横切处理程序（cross-cutting handler）。在第 14 章中，我们将更深入地介绍如何利用分离的请求和响应处理，构建响应式的客户端。

这里的重点是，在横切关注点的处理上，客户端应用程序的开发者不再受到 API 封装程序库的设计限制。API 提供者可以提供强类型的链接，只关注与其 API 相关的代码实现，而将通用的 HTTP 关注点留给通用的 HTTP 代码库处理。

4. 链接书签

使用链接生成 HttpRequestMessage 实例，有一个附带的好处是可以很方便地重复发起请求。一个 HttpRequestMessage 实例只能用于发送一个 HTTP 请求，而链接对象可以创建配置一次，用于发送多个请求。一个链接对象也可以修改一个或多个参数，创建一个新的请求。

链接也可以作为客户端状态的一部分进行存储，当作某种临时书签。当开发者首次接触超媒体 API 的时候，经常产生的一个不满和担心是，需要发出多个请求，才能从 API 的根资源访问到所需的资源。通过保存链接书签，客户端可以缓存链接进行重用，将额外的往复通信减到最少。

假设我们要给 Issue API 的根增加一个 json-home 文档，内容可能如下所示：

```
{
            "resources": {
            "http://example.org/rel/issue": {
            "href-template": "/example.org/issueApi/issue/{id}",
                     "href-vars": {
                       "id": "http://example.org/param/issueid"
                     }
            },
        "http://example.org/rel/issues": {
          "href": "/issueApi/issues",
        }

        "http://eample.org/rel/issueprocessor" : {
                         "href-template" : "issues/{id}/issueprocessor",
                         "href-vars": {
                      "id": "http://example.org/param/issueid"
                    }
        }
    }
}
```

对 API 根资源的初始请求可以获取这个主文档，解析链接，创建链接对象，然后将链接对象存储在一个全局可访问的字典中。例如：

```
var httpClient = new HttpClient();

var homeLink = new HomeLink() {
        Target = new Uri("http://example.org/issueApi")
}

var response = await httpClient.GetAsync(homeLink.CreateRequest());

var homedoc = homeLink.ParseHomeDocument(response, LinkFactory);

GlobalLinks = homeDoc.GetResourcesAsDictionary();

var issueLink = GlobalLinks["http://example.org/rel/issue"];
...
```

在这个简化的示例中，我们使用一个根文档，将发现的所有链接填充到 `GlobalLinks` 字典中。这段代码只在客户端应用程序启动时运行一次，支持超媒体的开销，就是客户端程序每运行一次都需要一次额外的往复通信。

将应用程序所有的链接都存储在一个根文档中，并不是超媒体 API 的最佳实践，因为这种做法不能根据上下文判断哪些链接可用。但是，每个 API 中都会有一些链接是在根文档中提供的。其他的链接可以通过系统中的其他资源发现得到，这些链接也可以保存为书签。

不存在一种适合所有 Web API 的解决方法。一些技术适用于某些情况，但并不一定适合所有情况。我们的目的是进行探索，研究什么技术可能解决追求可演化性带来的挑战。

创建一个发现资源的根文档，将所有链接进行全局缓存，这种做法虽然有其缺点，但是可演化性远好于在客户端程序库中硬编码 URI，而且实现起来非常简单。

9.2 应用程序工作流

使用链接对交互语义进行封装，提供一个间接层，使服务器可以演化其 URI 空间，这些帮助我们向消除分布组件之间耦合的目标迈进了一大步。

但是，客户端和服务器还是会由访问多个资源的交互协议联系在一起。如果一个客户端的代码逻辑为：获取资源 A，获取资源 B，展示信息，得到一些输入，然后把结果发送给资源 C。你必须修改客户端才能改变这个工作流。如果服务器在资源 A 和 B 之间加入一个额外的资源 A'，客户端不可能自动使用这个新资源。然而，如果资源 A 包含一个 `rel='next'` 的链接，那么客户端可以跟随这个 `netx` 链接，直到获得指向资源 C 的链接。如此一来，服务器可以选择添加或删除中间步骤，而不影响客户端的正常工作。

9.2.1 用户需知

将应用程序的工作流转移到服务器端，形成了这样一种客户端架构：对于如何根据宣称的媒体类型处理单独的资源表示，客户端表现得极为智能，并能基于链接关系的语义实现复杂的交互机制。但是，用户代理对于自己的行为在整个应用程序中的作用一无所知。这一特点简化了客户端代码，同时也有助于系统随着时间发生变化。使用同一个 Web 浏览器客户端的用户，可以先执行银行交易，又可以接着浏览菜谱网站，寻找晚餐灵感。

很多时候，客户端应用程序可能不希望将工作流控制完全移交给服务器。也许，一个客户端实际上与多个彼此无关的服务进行交互，或者客户端想实现服务器开发者没有考虑到的功能。即便是在这些情况下，适当放弃一些工作流控制也会给客户端带来益处。

要让服务器接管一些工作流的职责，一个方法是用应用程序域的目标来定义客户端行为，而不是用 HTTP 交互来定义。在传统的客户端应用程序中，应用程序方法和 HTTP 请求经常是一对一的关系。但是，实际满足用户的需求可能需要多个 HTTP 请求。如果对实现目标所需的交互进行封装，我们构建的工作单元在遇到变更时适应性就会更好。

假设有办法对实现用户目标的程序进行封装，我们会想使该目标的实现变得更加灵活。今天，实现用户目标也许需要两个 HTTP 请求，但是将来当服务器发现很多用户都会调用这个特殊的请求序列时，可能会优化 API，使其只用一个请求就实现目标。在理想情况下，客户端应该可以不进行修改就享受到这个优化。

为了获得这样的灵活性，我们需要对 HTTP 响应做出反应，而不是预期收到什么响应。标准的 HTTP 客户端程序库已经在一定程度上采取了这种做法。设想一个场景，客户端从资源 A 获取一个表示。服务器 API 有一些资源需要认证，其他资源不需要认证。`HttpClient` 持有一组凭据，但是在访问资源 A 时并不使用这些凭据，因为资源 A 不需要认证。由于某些外部因素，服务器决定改变行为，要求对资源 A 进行认证。当客户端试图获取资源 A 时，就会收到一个 `401 error` 和一个 `www-authenticate` 标头。客户端明白了问题所在，对此做出反应，使用凭据重新向资源 A 发送请求。对于发起 HTTP 请求的客户端程序来说，这一系列交互的发生可能都是完全透明的。

有些 HTTP 客户端在收到重定向（`3XX`）状态码时也会做出同样的反应。HTTP 程序库负责将单个请求转换成多个请求，以实现原本的目标。

把同样的想法在应用程序域中实现，我们就可以获得类似的灵活性，处理很多以前认为是破坏性的变更。

请看下面的代码片段，这可能是一个客户端应用程序的一部分：

```
public void SelectIssue(IssueLink issueLink) {

    var response = await _httpClient.SendAsync(issueLink.CreateRequest());
```

```
        var issue = issueLink.ParseResponse(response);

        var form = new IssueForm();
        form.Display(issue);
}
```

再请看下面的代码：

```
public void Select(Link aLink) {

        var response = await _httpClient.SendAsync(aLink.CreateRequest());
        List<Issue> issues = new List<Issue>();

        switch(response.Content.Headers.ContentType.MediaType) {

                case "application/collection+json" :
                        LoadIssues(issues, response.Content)
                        break;

                case "application/issue+json" :
                        issues.Add(issueLink.ParseResponse(response));
                        break;
        }

        foreach(var issues in issues) {
                var form = IssueFormFactory.CreateIssueForm();
                form.Display(issue);
    }
}
```

这是一个虚构的示例，简单演示可能的实现。在这个示例中，我们认识到请求可能返回不同的媒体类型。如果请求的链接只返回一个问题，我们就直接显示这个问题；如果请求返回一个问题集合，那么就搜索问题链接，加载整个问题集合，全部予以显示。

如果你能够实现下面的代码，客户端应用程序将会变得非常有意思：

```
public void Select(Link aLink) {

        var response = await _httpClient.SendAsync(aLink.CreateRequest());
        GlobalHandler.HandleResponse(aLink, response);
}
```

使用这段代码中的方法，客户端应用程序和用于获取响应的链接，以及用户处理响应消息的具体响应消息，这三者中的上下文是一样多的。这种方法实现了工作流去耦的终极目标，和 Web 浏览器的工作方式非常类似。

1. 处理所有版本

当服务器对其 API 进行优化时，客户端很有可能不具备利用这些优化的必要知识。只要服务器仍然保留未优化的交互机制，客户端就能继续工作，浑然不知还有更快的方法可用。在客户端的下一个版本中，可以引入代码，一旦存在优化方式就加以利用。关键在于客户

端要进行必要的功能检测，判断功能是否可用。如果服务器 API 只有一个实例，如 Twitter 或 Facebook，这可能并不是个问题。但是，如果为如 WordPress 这样的产品开发客户端程序库，你就不能保证服务器 API 的哪些功能是可用的。

将反应性行为与功能检测相结合，客户端和服务器就可以持续合作，无需进行版本号协调。这种功能应该从一开始就进行计划，不太容易添加到一个已经存在的客户端中。

2. 变更不可避免

在不同的情况下，服务器可能做出客户端可适应的变更。我们将讨论一些经常发生的情况。

通过分析客户端访问 API 的路径，服务器可能决定引入一个快捷链接，绕过一些中间表示。有时，服务器会给 URI 模板添加一个参数，实现这种快捷访问。客户端可以通过链接关系寻找这种快捷链接，如果快捷链接不存在，客户端可以继续使用原来的方式进行访问。

API 可能引入效率更高，或者携带更多语义的新的表示。新的媒体类型总是不断出现。如果一个客户端实现支持这种新格式，就可以在请求的 `Accept` 标头中加入这个类型，告诉服务器自己支持这种新格式。为了与旧版本的 API 兼容，客户端仍然需要支持旧格式。假设客户端在设计时就知道服务器可能返回多种格式，通常很容易就能实现这种功能。

在设计表示时，服务器需要决定嵌入相关资源的信息，或是提供相关资源的链接。有时，服务器需要修改已经做出的决定。如果一个资源表示规模过大，嵌入资源可能需要替换为链接。如果客户端总是访问某些相关资源的链接，那么在返回时嵌入这些内容可能效率会更高。如果客户端设计为在需要时可以透明地嵌入链接资源，那么服务器就可以修改资源表示，而不会破坏客户端。

人工驱动（human-driven）的客户端经常需要向用户展示一组可用的资源链接。在这种情况下，遍历页面上的所有链接并逐一展示，比将静态 UI 元素绑定到已知链接要好。采用动态构建 UI 的方法，当服务器可以添加资源链接时，客户端能够自动访问这些链接。

如果资源链接移除，即便这些链接绑定到固定的 UI 元素，客户端也可以使这些元素失效，说明相应的资源不可用。客户端不应该仅因一个链接不存在就不工作。有时，移除一个链接可能使用户无法完成一个特定功能，但我们可以认为其他的功能依然可用。对于客户端来说，移除一个功能不应该是破坏性的变更。

由于 URI 空间重组，或者将资源移到新主机上以均衡负载，服务器可能会改变资源到新的 URI。客户端应该能够透明地处理由此导致的重定向请求。

服务器每次向链接添加新的查询参数时，都应该给这些新增的参数提供默认值。如果服务器没有给新增参数提供默认值，就会造成破坏性变更，在这种情况下，服务器应该添加一个新的链接和链接类型，定义这个新的需求。客户端应用程序应该能够继续安全地使用链

接，无需指定新的参数。客户端的未来更新应该能够使用这个新参数。

如果服务器发现不再需要标识资源的一个参数，就应该更新 URI 模板，不再包含这个参数值。当客户端试图设置不在 URI 模板中的参数时，不应该得到失败结果。解析后得到的 URL 应当不再包含已经移除的参数，否则服务器可能返回状态码 `404`。

在一些情况下，以前接受 `GET` 和 `POST` 方法的资源可能会进行拆分，单独创建一个资源处理 `POST` 请求。在这种情况下，服务器应该提供另一个链接和链接关系，并实现从原有资源 `POST` 的重定向，以处理过渡时期的请求。

一个交互序列可能添加新的步骤。在这种情况下，服务器可以将新步骤中的一个链接标记为“默认的”链接。在不知道如何处理一个特定表示时，客户端应该设计为使用默认链接。

链接提示 IETF 互联网草案（参见 http://tools.ietf.org/html/draft-nottingham-link-hint-00）引入了对链接使用“弃用”（deprecated）属性进行修饰的功能。这个属性可以告知客户端，未来这个链接不再受到支持。除此之外，服务器开发者应该记录这些弃用链接的使用情况，以及访问这个资源的用户代理。基于这些信息，我们可以进行离线沟通，敦促客户端开发者停止使用这些弃用链接。

毫无疑问，服务器 API 可能发生很多别的变更情况。但是，这些例子说明，Web 架构的设计方式使得客户端可以适应这些变更。如果我们不再费力地进行客户端和服务器的版本兼容管理，而是把精力放在处理 API 的可能变更上，那么就可以更快进行演化，使用户满意度更高。

9.2.2 带有使命的客户端

在完全由人工驱动的使用体验（如 Web 浏览器）中，交互模型似乎是 1:1 的，其实并非如此。点击一个链接会加载一个 HTML 页面，但是，浏览器需要加载链接的样式表、图像和脚本。只有当所有这些请求都完成时，展示页面的目标才算是完成了。

要描述这种交互的封装，我能想到的最好的词是*使命*（mission）。一个使命是实现一个客户端目标所需的请求和响应处理的具体实现。这个词虽然听起来有点怪，但其好处是避免了过载其他的软件术语，如*任务*（task）或*事务*（transaction）。使命这个词还强调了目标的重要性，而非具体实现的细节。而且，人们经常将 HTTP 客户端称为*代理*（agent），代理和使命这两个词刚好相当契合。

之前我们讨论过，链接关系可以用于标识一个 HTTP 交互的语义。有时候链接关系也能传达多个交互的需求。链接关系查询就是一个例子。一个查询链接指向一个 `OpenSearchDescription` 文档，该文档包含了关于如何对一个网站或 API 的资源进行查询的信息。最简单的情况下，这个描述文档包含一个 URL 模板，其中的 `{searchTerms}` 符号可以替换为客户端的实际查询词。

接下来的类展示了如何创建一个使命，对一系列行为进行封装：得到 OpenSearchDocument 文档、解释文档、构建 URL、执行搜索和返回搜索结果。

```
public class SearchMission
    {
        private readonly HttpClient _httpClient;
        private readonly SearchLink _link;

        public SearchMission(HttpClient httpClient, SearchLink link)
        {
            _httpClient = httpClient;
            _link = link;
        }

        public async Task<HttpResponseMessage> GoAsync(string param)
        {
            var openSearchDescription = await LoadOpenSearchDescription();
            var link = openSearchDescription.Url;
            link.SetParameter("searchTerms", param);
            return await _httpClient.SendAsync(link.CreateRequest());
        }

        private async Task<OpenSearchDescription> LoadOpenSearchDescription()
        {
            var response = await _httpClient.SendAsync(_link.CreateRequest());
            var desc = await response.Content.ReadAsStreamAsync();
            return new OpenSearchDescription(
                        response.Content.Headers.ContentType, desc);
        }
    }
```

这个使命类自身并不包括解释 OpenSearchDescription 文档的细节，这部分由媒体类型解析库完成。使命只关注交互的协调。一个 SearchMission 对象可以执行多个搜索。

使命可以是完全与 HTTP 资源进行的基于算法的交互。一个客户端应用程序初始化一个使命，然后持续执行，直到目标实现，或者使命失败。使命可以成为一个重用单元，多个使命可以结合起来实现更大的目标。

使命也可以是交互式过程，在一些交互之后，控制权会交还给人工用户，等待用户下一步的指令。为了实现这个功能，使命需要设计有接到用户界面层的某种接口。使命必须能够使用这个接口，将当前的状态展示给用户，并接受下一步的指令。这个用户界面层必须向人类提供一组可供选择的链接，然后将选中的链接传回给使命。

接下来的示例包括一个非常简单的交互式使命，以及一个用作超媒体 REPL（Read-Eval-Print Loop，“读取 – 求值 – 输出”循环）的小型控制台应用程序。这个客户端应用程序必须使用一个链接调用使命的 GoAsync 方法。使命会使用这个链接，获取返回的资源表示中的任何链接。一个简单的控制台应用程序使用一个循环，向用户展示初始的资源表示，并让用户输入另一个链接名以选择使用这个链接。客户端随后会要求使命使用这个链接，并

重解析返回的链接。

```
public class ExploreMission
{
    private readonly HttpClient _httpClient;

    public Link ContextLink { get; set; }
    public HttpContent CurrentRepresentation { get; set; }
    public Dictionary<string,Link> AvailableLinks { get; set; }

    public ExploreMission(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task GoAsync(Link link)
    {
        var response = await _httpClient.SendAsync(link.CreateRequest());
        if (response.IsSuccessStatusCode)
        {
            ContextLink = link;
            CurrentRepresentation = response.Content;
            AvailableLinks = ParseLinks(CurrentRepresentation);
        }
    }

    private Dictionary<string,Link>
            ParseLinks(HttpContent currentRepresentation)
    {
        // 根据返回的媒体类型解析表示中的链接
    }
}

static void Main(string[] args)
{
    var exploreMission = new ExploreMission(new HttpClient());

    var link = new Link() {Target = new Uri("http://localhost:8080/")};
    string input = null;
    while (input != "exit")
    {
        exploreMission.GoAsync(link).Wait();
        Console.WriteLine(exploreMission.CurrentRepresentation
                    .ReadAsStringAsync().Result);

        Console.Write("Enter link relation to follow link : ");
        input = Console.ReadLine();
        link = exploreMission.AvailableLinks[input];
    }
}
```

一个有实际用途的交互式客户端应用程序所做的工作，显然要比这个示例多得多。但是，基本前提是一样的：使用一个链接，更新客户端状态，向用户展示可用链接，让用户选择链接，然后重复这一过程。

9.2.3 客户端状态

客户端状态，也经常称为*客户端应用程序状态*，是客户端应用程序当前跟踪的所有执行使命的聚合。一个 Web 浏览器包含一组*浏览上下文*（browsing context，参见 http://www.w3.org/TR/html5/browsers.html#windows），每个上下文都可以算是一个顶级使命（top-level mission）。

如果构建的客户端应用程序框架能够管理一组活动的使命，你就可以把问题分解为开发面向目标的使命，这些使用可以用应用域的概念来描述。

虽然客户端应用程序状态由一组活动使命组成，但是，每个使命在执行中都必须谨慎地处理状态的累积。

让我们回到 `SearchMission` 的示例，`SearchMission` 对象可以保存 `OpenSearchDescription` 对象的一个引用，进行重用。但是，一旦客户端采取这种做法，就接管了这个资源表示的生存期的所有权，不再由服务器控制。理想情况下，服务器会提供缓存指令，使 `OpenSearchDescription` 资源表示能在本地缓存很长一段时间，重复使用 `SearchMission` 对象时，请求描述文档就不会引起网络往复通信。客户端也不必管理和共享 `SearchMission` 对象的引用，因为本地 HTTP 缓存是持久的，客户端应用程序可以多次执行重用已保存的 `OpenSearchDocument` 文档。

对于很多人来说，这种避免保存客户端状态的做法有悖常理。在构建客户端应用程序时，传统的经验是，如果我们保存从远程服务器得到的状态，就可以避免在之后发起网络往复通信。客户端管理资源状态生存期的问题是，你最终会得到很多随意的缓存机制。通常，这些客户端缓存机制比 HTTP 提供的缓存机制要简易得多。客户端通常只支持两种生存期的范围：由应用程序生存期决定的全局范围，以及某种*工作单元*范围。客户端缓存机制没有缓存过期的概念，当然也没有等效于条件 `GET` 的操作。如果把大多数的客户端缓存交给 HTTP 缓存机制，客户端代码可以更为简单；服务器指定资源生存期，可以减少一致性问题；没有那么多上下文信息影响客户端对服务器响应做出反应，调试工作也会变得更简单。

9.3 小结

虽然 Web 进入我们的生活已经快 20 年了，但是，在构建适合 Web 架构的客户端方面，我们的经验却依然十分有限。真正的 Web 客户端主要局限于 Web 浏览器、RSS 摘要阅读器和网络爬虫。拥有开发其中任意一类工具经验的开发者寥寥无几。最近开始流行的单页应用程序（single-page application）带来了一种新的模式，人们尝试在一个用户代理内部构建一个用户代理，这种模式具有其独特的挑战。

依赖于分布式服务的本地“应用”的兴起，使人们对构建 Web 客户端重新产生了兴趣。第一批这种分布式应用试图复制 20 世纪 90 年代的客户端 / 服务器架构。但是，要构建出像

Web 浏览器一样，真正利用 Web 功能的应用，我们需要模拟 Web 浏览器的一些架构特征。

使用本章中讨论的技术，开发者构建的客户端将会更加持久，更少出错，性能更好，对网络失败的容忍度更高。遗憾的是，时至今日，只有很少的支持程序库能帮助我们使用这些技术。构建能够演化的松耦合客户端会带来很多的好处，随着更多的开发者开始理解这些好处，未来可望出现更多的可用工具。

作为实践这些技术的第一步，当你编写发出网络请求的客户端代码时，请停下来，想一想，然后问自己：如果事情按照预期的情况发生，会怎么样；如果事情不按预期发生，又会怎么样；HTTP 能怎样帮助我处理这些情况。

第三部分

Web API细节





第10章
HTTP编程模型

消息，完整的消息，只有这个消息。

本章将介绍新的 .NET 框架 HTTP 编程模型，这个编程模型是 ASP.NET Web API 和新的客户端 HTTP 支持（特别是 `HttpClient` 类）的核心。这个模型包含在 .NET 4.5 中，但是也可以通过 NuGet 包（http://www.nuget.org/packages/Microsoft.Net.Http/2.0.20710）获得 .NET 4.0 的版本。新的 HTTP 编程模型定义了一个新的程序集——`System.Net.Http.dll`，对主要的 HTTP 概念（即请求和响应消息、标头以及正文内容）进行了具有类型地编程抽象。

新的 HTTP 编程模型有一个辅助程序集 `System.Net.Http.Formatting.dll`，其中引入了媒体类型格式化程序的概念（第 13 章将对此进行介绍），以及一些工具扩展方法和定制的 HTTP 内容类型。`System.Net.Http.Formatting.dll` 可以从“Microsoft ASP.NET Web API Client Libraries”（参见 http://www.nuget.org/packages/Microsoft.AspNet.WebApi.Client）NuGet 包下载，源代码属于 ASP.NET 项目（参见 https://aspnetwebstack.codeplex.com/）。虽然这个软件包名称是“客户端代码库”，但实际上在客户端和服务器端都可以使用。本章，我们将描述 `System.Net.dll` 和 `System.Net.Http.Formatting.dll` 的功能，并不特意区分二者。

在引入这个新模型前，.NET 框架已经包含了多个用于处理 HTTP 概念的编程模型。在客户端，`System.Net.HttpWebRequest` 类可以用于初始化 HTTP 请求，处理相关的响应；在服务器端，`System.Web.HttpContext` 及其相关类（如 `HttpRequest` 和 `HttpResponse`）用在 ASP.NET 上下文中，代表单个请求和响应。在服务器端，还有 `System.Net.HttpListenerContext` 类，用在自托管的 `System.Net.HttpListenser` 中，提供对 HTTP 请求和响应对象的访问。

遗憾的是，所有这些编程模型都存在若干问题。新的模型，即 `System.Net.Http` 编程模型，致力于解决这些问题，具有如下特点：

- 在客户端和服务器端使用同样的类；
- 基于新的 TAP（Task Asynchronous Pattern，任务异步模式），而非旧的 APM（Asynchronous Programming Model，异步编程模型），也就是说，可以使用 .NET 4.5 中提供的 `async` 和 `await`；
- 易于在测试场景中使用；
- 对 HTTP 消息的表示*类型更强*——换句话说，就是将 HTTP 标头值表示为类型，而不是松散的字符串字典；
- 更加忠于 HTTP 规范，即不在 HTTP 协议上堆加不同的抽象层；
- 将较新的版本打包为可移植的类库，可以在多种平台上使用。

在接下来的小节中，随着我们更详细地介绍这个新的模型，以上这些特点会变得更加清晰。我们将首先介绍表示基础 HTTP 概念（也就是请求和响应消息）的类型；接着将展示如何通过一组具体的类，对请求和响应消息以及内容标头进行表示和处理；最后讨论如何生成和使用消息有效载荷的内容。

在我们开始之前，请注意：旧的 HTTP 编程模型仍可使用，并继续受到支持。例如，ASP.NET 管道依然基于旧的 `System.Net.HttpWebRequest`。

10.1 消息

第 1 章介绍过，HTTP 协议的工作方式是在客户端和服务器之间交换请求和响应消息。很自然地，HTTP 编程模型的核心就是消息抽象，表示为两个具体类：`HttpRequestMessage` 和 `HttpResponseMessage`。这两个类位于新的 `System.Net.Http` 命名空间，如图 10-1 所示。`HttpRequestMessage` 和 `HttpResponseMessage` 都包含：

- 一个*起始线*（start line）；
- 一列*标头字段*（header field）；
- 一个可选的有效载荷正文（payload body）。

对于请求消息，起始线由如下 `HttpRequestMessage` 属性表示：

- 请求的方法（如 `GET` 或 `POST`），定义了这个请求的目的；
- `RequestUri`，标识目标资源；
- 协议版本 `Version`（如 `1.1`）。

对于响应消息，起始线由如下 `HttpResponseMessage` 属性表示：

- 协议版本 `Version`（如 `1.1`）；
- 请求的状态码 `StatusCode`（一个三位的整数），以及提示信息 `Reason Phrase` 字符串。

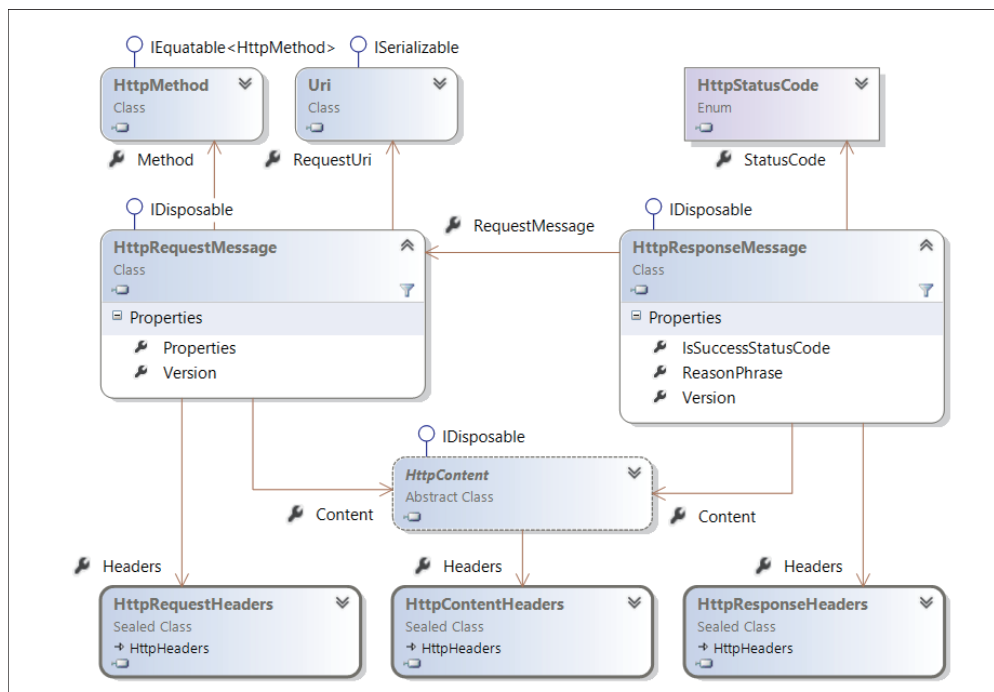


图 10-1：`HttpRequestMessage` 和 `HttpResponseMessage` 类

响应消息还包含一个 `RequestMessage` 属性，指向对应的请求消息。

请求和响应消息都可以包含一个可选的消息正文，表示为 `Content` 属性。在 10.3 节中，我们将详细介绍消息内容如何表示、创建和使用，还将描述基于 `HttpContent` 的类层次结构。

这两种消息类型以及消息内容，都可以使用相应的标头，携带元数据。10.2 节将介绍处理这些标头的编程模型。

`HttpRequestMessage` 和 `HttpResponseMessage` 类都不是抽象类，可以在用户代码中很容易地实例化，如下所示：

```
[Fact]
public void HttpRequestMessage_is_easy_to_instantiate()
{
    var request = new HttpRequestMessage(
        HttpMethod.Get,
        new Uri("http://www.ietf.org/rfc/rfc2616.txt"));

    Assert.Equal(HttpMethod.Get, request.Method);
    Assert.Equal(
        "http://www.ietf.org/rfc/rfc2616.txt",
        request.RequestUri.ToString());
    Assert.Equal(new Version(1,1), request.Version);
}
```

```
[Fact]
public void HttpResponseMessage_is_easy_to_instantiate()
{
    var response = new HttpResponseMessage(HttpStatusCode.OK);
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
    Assert.Equal(new Version(1,1), response.Version);
}
```

消息类很容易在测试场景中使用，与其他用于表示同样概念的 .NET 框架类形成强烈对比。

- System.Web.HttpRequest 类，在 ASP.NET 的 System.Web.HttpContext 中表示一个请求。这个类有公共的构造函数，但是只能由基础结构使用。
- System.Web.HttpRequestBase 类，在 ASP.NET MVC 中使用。这个类是抽象类，无法直接初始化。
- System.Net.HttpWebRequest 类，用于在客户端表示 HTTP 请求。这个类有公共的构造函数，但是构造函数都已过时，应该通过 WebRequest.Create 工厂方法进行实例化。

HttpRequestMessage 和 HttpResponseMessage 类都只表示 HTTP 消息，并不包含其他的上下文属性，因此既可用于客户端，又可用于服务器端。这一点与其他的 HTTP 类不同，例如，ASP.NET 的 HttpRequest 类包含一个属性，用于保存服务器上的虚拟应用程序根路径，这个属性对客户端显然并不适用。

HttpMethod 实例包含方法字符串（如 GET 或 POST），代表请求方法。HttpMethod 类还包含一组静态属性，对应 RFC 2616 中定义的方法。

```
public class HttpMethod : IEquatable<HttpMethod>
{
       public string Method {get;}
       public HttpMethod(string method);

       public static HttpMethod Get {get;}
       public static HttpMethod Put {get;}
       public static HttpMethod Post {get;}
       public static HttpMethod Delete {get;}
       public static HttpMethod Head {get;}
       public static HttpMethod Options {get;}
       public static HttpMethod Trace {get;}
}
```

要使用一个新方法，如 RFC 5789 中定义的 PATCH，我们必须明确使用方法字符串，对 HttpMethod 进行初始化，代码如下所示：

```
[Fact]
public async Task New_HTTP_methods_can_be_used()
{
    var request = new HttpRequestMessage(
        new HttpMethod("PATCH"),
```

```
            new Uri("http://www.ietf.org/rfc/rfc2616.txt"));
        using(var client = new HttpClient())
        {
            var resp = await client.SendAsync(request);
            Assert.Equal(HttpStatusCode.MethodNotAllowed, resp.StatusCode);
        }
    }
```

枚举类型 `HttpStatusCode` 表示响应的状态码，其中包含了 HTTP 规范定义的所有状态码：

```
public enum HttpStatusCode
  {
    Continue = 100,
    SwitchingProtocols = 101,
    OK = 200,
    Created = 201,
    Accepted = 202,
    ...
    MovedPermanently = 301,
    Found = 302,
    SeeOther = 303,
    NotModified = 304,
    ...
    BadRequest = 400,
    Unauthorized = 401,
    ...
    InternalServerError = 500,
    ...
}
```

我们也可以把整数转换为 `HttpStatusCode` 类型，得到新的状态码：

```
[Fact]
public void New_status_codes_can_also_be_used()
{
    var response = new HttpResponseMessage((HttpStatusCode) 418)
                       {
                          ReasonPhrase = "I'm a teapot"
                       };
    Assert.Equal(418, (int)response.StatusCode);
}
```

`HttpRequestMessage` 还包含一个 `Properties` 属性：

```
public IDictionary<string, Object> Properties { get; }
```

当消息在服务器或者客户端本地进行处理时，`Properties` 属性用于保存附加的消息信息。例如，`Properties` 属性可以保存处理栈底层（例如消息处理程序）产生的信息，供上层使用（例如控制器）。

`Properties` 属性并不属于任何标准的 HTTP 消息，当消息进行序列化，准备传输时，不会保留 `Properties` 属性。`Properties` 属性只是一个通用的容器，保存本地消息属性，例如：

- 与接受消息的连接相关的客户端认证；
- 将消息与配置路由进行匹配，得到的路由数据。

这些属性存储在一个字典表中，对应于字符串键值。HttpPropertyKeys 类定义了一组常用的键。通常情况下，这些消息属性可以通过扩展方法访问，如 System.Net.Http.HttpRequestMessageExtensions 类中定义的方法。

```
public static IHttpRouteData GetRouteData(this HttpRequestMessage request)
{
  if (request == null)
     throw System.Web.Http.Error.ArgumentNull("request");
  else
     return HttpRequestMessageExtensions.GetProperty<IHttpRouteData>(
                              request, HttpPropertyKeys.HttpRouteDataKey);
}
```

Web API v2 中引入的 HttpRequestContext 类，也是由底层的托管层附加到请求属性，供上层使用的信息。

```
public static HttpRequestContext
    GetRequestContext(this HttpRequestMessage request)
{
    ...
    return request.GetProperty<HttpRequestContext>(
        HttpPropertyKeys.RequestContextKey);
}

public static void
    SetRequestContext(this HttpRequestMessage request,
                      HttpRequestContext context)
{
    ...
    request.Properties[HttpPropertyKeys.RequestContextKey] = context;
}
```

也就是说，HttpRequestContext 类将一组属性（例如：客户端证书，或者请求者的身份信息）聚合在一个具有类型的模型中。

```
public class HttpRequestContext
{
    public virtual X509Certificate2 ClientCertificate { get; set; }
    public virtual IPrincipal Principal { get; set; }
    // ...
}
```

10.2 标头

在 HTTP 中，请求和响应消息，以及消息内容自身，都可以使用称为标头（header）的额外字段，包含更多信息。例如：

- User-Agent 标头字段，为请求提供扩展信息，描述产生这个请求的应用程序；
- Server 标头字段，为响应提供关于源服务器软件的扩展信息；
- Content-Type 标头字段，定义请求或响应有效载荷正文中，资源表示使用的媒体类型。

每个标头都包括一个名字和一个值，这个值可以是列表。HTTP 规范允许一个消息的多个标头使用同样的名字。但是，HTTP 规范也规定，多个标头使用同一名字，效果等同于定义一个标头，其值为多个标头值的合集。已注册的 HTTP 标头（参见 http://www.iana.org/assignments/message-headers/message-headers.xml）由 IANA 维护。

如图 10-1 所示，请求和响应消息类都包含一个 Headers 属性，指向一个具有类型的标头容器类。但是，内容标头（如 Content-Type）不属于请求或响应的标头集合。内容标头属于内容标头集合，可以通过 HttpContent 的 Headers 属性访问。

```
[Fact]
public async void Message_and_content_headers_are_not_in_same_coll()
{
    using(var client = new HttpClient())
    {
        var response = await client
            .GetAsync("http://tools.ietf.org/html/rfc2616");
        var request = response.RequestMessage;
        Assert.Equal("tools.ietf.org",request.Headers.Host);
        Assert.NotNull(response.Headers.Server);
        Assert.Equal("text/html",
            response.Content.Headers.ContentType.MediaType);
    }
}
```

请注意，Server 标头在 response.Headers 容器中，而 Content Type 标头在 response.Content.Headers 容器中。

HTTP 编程模型为这三种标头上下文，各自定义了标头容器类：

- HttpRequestHeaders 类包含请求标头；
- HttpResponseHeaders 类包含响应标头；
- HttpContentHeaders 类包含内容标头。

这三个类都有一组属性，以强类型方式提供标准标头的访问。例如：HttpRequestHeaders 类包含一个 Accept 属性，其类型为 MediaTypeWithQualityHeaderValue 集合，集合中每个条目包含如下内容：

- MediaType 字符串属性，其值为媒体类型标识符（例如：application/xml）；
- Quality 属性（如 0.9）；
- CharSet 字符串属性；
- Parameters 集合属性；

下一段代码展示了 Accpet 标头的使用。因为类模型提供了对所有组成部分（如质量参数和字符集）的访问，使用标头非常容易。

```
[Fact]
public void Classes_expose_headers_in_a_strongly_typed_way()
{
    var request = new HttpRequestMessage();
    request.Headers.Add(
        "Accept",
        "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8");

    HttpHeaderValueCollection<MediaTypeWithQualityHeaderValue> accept =
        request.Headers.Accept;
    Assert.Equal(4,accept.Count);

    MediaTypeWithQualityHeaderValue third = accept.Skip(2).First();
    Assert.Equal("application/xml", third.MediaType);
    Assert.Equal(0.9, third.Quality);
    Assert.Null(third.CharSet);
    Assert.Equal(1,third.Parameters.Count);
    Assert.Equal("q",third.Parameters.First().Name);
    Assert.Equal("0.9", third.Parameters.First().Value);
}
```

这一功能极大地简化了标头的生成和使用，将有时略显烦琐的 HTTP 语法规则进行了抽象。这些属性也可以用于轻松构建标头的值：

```
[Fact]
public void Properties_simplify_header_construction()
{
    var response = new HttpResponseMessage();
    response.Headers.Date =
        new DateTimeOffset(2013,1,1,0,0,0, TimeSpan.FromHours(0));
    response.Headers.CacheControl = new CacheControlHeaderValue
    {
        MaxAge = TimeSpan.FromMinutes(1),
        Private = true
    };

    var dateValue = response.Headers.First(h => h.Key == "Date")
        .Value.First();
    Assert.Equal("Tue, 01 Jan 2013 00:00:00 GMT", dateValue);

    var cacheControlValue = response.Headers
        .First(h => h.Key == "Cache-Control").Value.First();
    Assert.Equal("max-age=60, private", cacheControlValue);
}
```

请注意，CacheControlHeaderValue 类为每个 HTTP 缓存指令（如 MaxAge 和 Private）提供一个属性。Date 标头是用一个 DateTimeOffset 值创建的，而不是使用字符串，简化了构建格式正确的标头值的过程。

有些标头的值是纯量的（如 Date），可以直接赋值；其他的标头是集合值，用 HttpHeaderValueCollection<T> 泛型类表示，可以增加和删除集合中的值：

```
request.Headers.Date = DateTimeOffset.UtcNow;
request.Headers.Accept.Add(new MediaTypeWithQualityHeaderValue("text/html",1.0));
```

图 10-2 展示了三个标头容器类，对应三种标头上下文。这些类没有公共的构造函数，不能简单地单独创建，而是在消息或内容实例创建时一同创建。

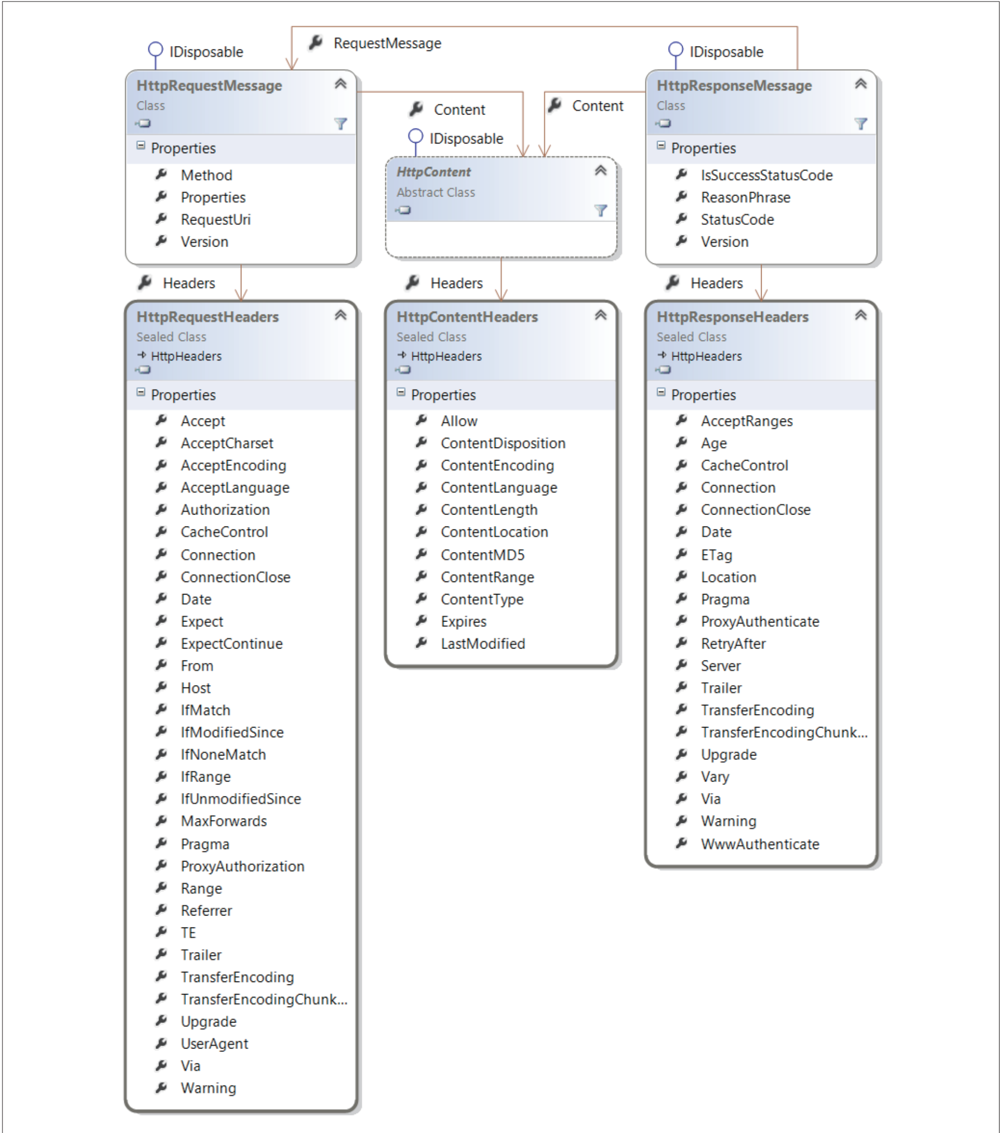


图 10-2：三个标头容器类

这三个类提供的属性只限于由 HTTP RPC 定义的标头。例如：HttpRequestHeader 类包含的属性，只对应于 HTTP 请求可以使用的标头。而且，HttpRequestHeader 类不提供添加非标准标头的方法。但是，这三个类都派生自 HttpHeaders 抽象类（如图 10-3 所示），这个抽象类提供一组方法，可以对标头进行较低层次的访问。

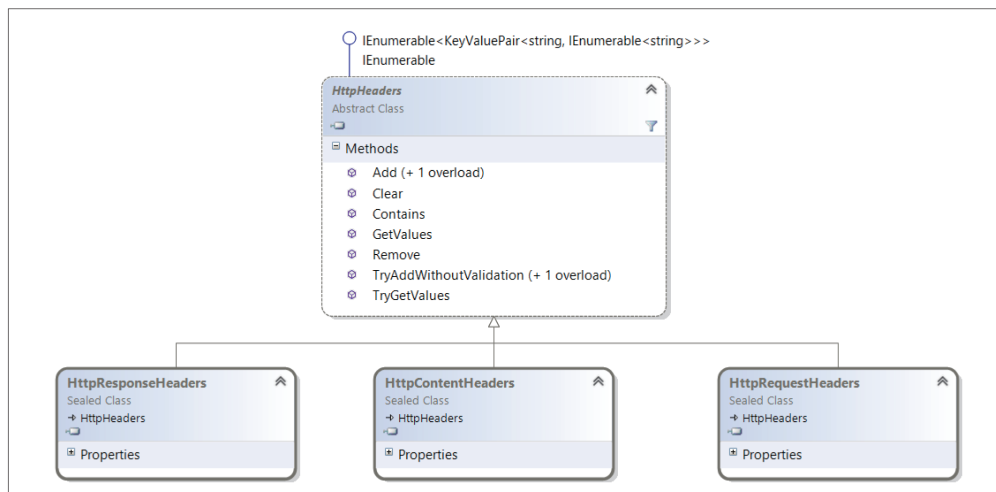


图 10-3：基类 HttpReaders 提供对标头集合的无类型访问

首先，HttpReaders 类实现了如下接口：

```
IEnumerable<KeyValuePair<string,IEnumerable<string>>>
```

使用这个接口，我们可以将所有标头当作值对序列访问，值对中标头名是一个字符串，标头值是一个字符串序列。这个接口保留了标头的顺序，支持了标头值作为列表的情况。HttpHeaders 了也包含一组方法，可以添加和删除标头。

Add 方法可以向容器添加标头。如果要添加的标头有标准名，在添加之前标头值会进行验证。Add 方法还会验证标头是否可以有多个值。

```
[Fact]
public void Add_validates_value_domain_for_std_headers()
{
    var request = new HttpRequestMessage();
    Assert.Throws<FormatException>(() =>
        request.Headers.Add("Date", "invalid-date"));
    request.Headers.Add("Strict-Transport-Security", "invalid ;; value");
}
```

另一方面，TryAddWithoutValidation 方法不执行标头值检验。但是，如果值是无效的，就不能通过有类型的属性进行访问。

```
[Fact]
public async void
    TryAddWithoutValidation_doesnt_validates_the_value_but_preserves_it()
{
    var request = new HttpRequestMessage();
    Assert.True(request.Headers
        .TryAddWithoutValidation("Date", "invalid-date"));
    Assert.Equal(null, request.Headers.Date);
    Assert.Equal("invalid-date", request.Headers.GetValues("Date").First());

    var content = new HttpMessageContent(request);
    var s = await content.ReadAsStringAsync();
    Assert.True(s.Contains("Date: invalid-date"));
}
```

了解了消息和内容如何使用标头扩展信息，下一节我们将关注内容自身。

10.3 消息内容

在新的 HTTP 编程模型中，HTTP 消息的正文由抽象基类 `HttpContent` 表示（参见图 10-4）。`HttpRequestMessage` 和 `HttpResponseMessage` 都包含一个 `HttpContent` 类型的 `Content` 属性（参见图 10-1）。

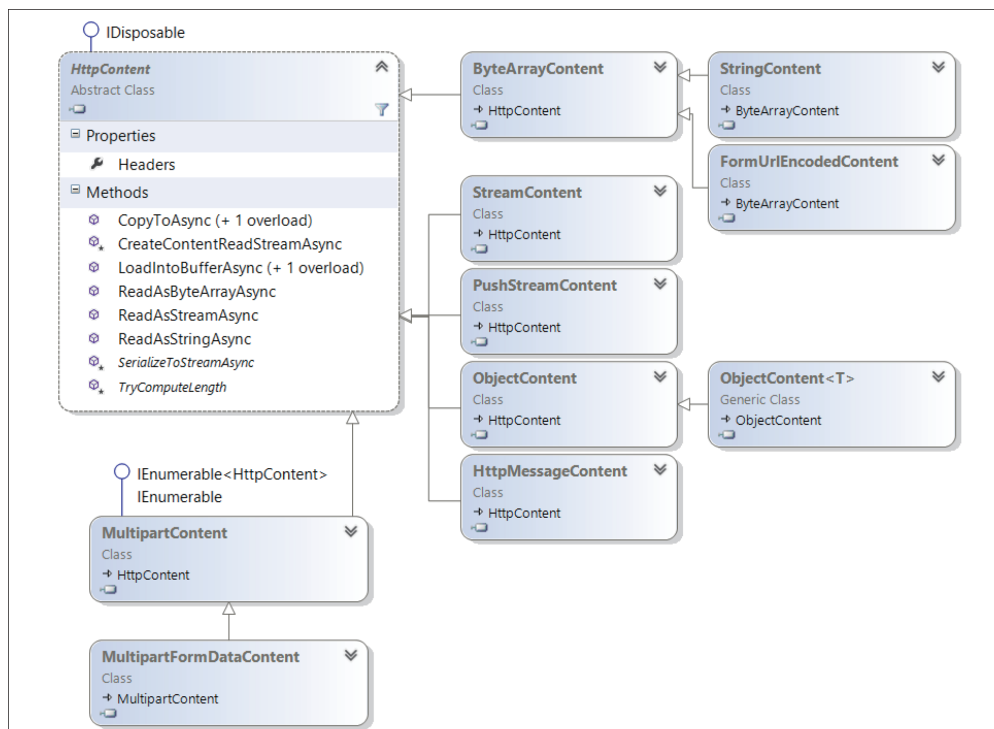


图 10-4：`HttpContent` 基类以及相关类层次

在这一节中，我们将了解：

- 如何通过 HttpContent 的方法使用消息内容；
- 如何通过现有的 HttpContent 派生具体的类，或者创建一个新类，生成消息内容。

10.3.1　使用消息内容

在创建消息内容时，我们可以选择一个可用的由 HttpContent 派生的具体类。然而，在使用消息内容时，我们只能使用 HttpContent 的方法或者扩展方法。

除了前一节中介绍的 Headers 属性，HttpContent 还包含如下非虚拟的公共方法：

- Task CopyToAsync(Stream, TransportContext)
- Task<Stream> ReadAsStreamAsync()
- Task<string> ReadAsStringAsync()
- Task<byte[]> ReadAsByteArrayAsync()

第一方法可以用于以*推送*（push）方式访问原始的消息内容。我们将一个流传递给 CopyToAsync 方法，然后 CopyToAsync 把消息内容写入（推送）到这个流中。返回的 Task 可以用于与复制终止进行同步：

```
[Fact]
public async Task HttpContent_can_be_consumed_in_push_style()
{
    using (var client = new HttpClient())
    {
        var response =
            await client.GetAsync("http://www.ietf.org/rfc/rfc2616.txt",
            HttpCompletionOption.ResponseHeadersRead
            );
        response.EnsureSuccessStatusCode();
        var ms = new MemoryStream();
        await response.Content.CopyToAsync(ms);
        Assert.True(ms.Length > 0);
    }
}
```

前面这个示例使用了 HttpCompletionOption.ResponseHeadersRead 选项，使 GetAsync 方法在响应头读取之后立即终止，响应内容可以使用 CopyToAsync 方法访问，无需进行缓冲。

你也可以使用 ReadAsStreamAsync 方法，以*拉取*（pull）方式访问原始的消息内容。这个方法异步返回一个流，可以从中获取内容。

```
[Fact]
public async Task HttpContent_can_be_consumed_in_pull_style()
{
    using (var client = new HttpClient())
```

```
    {
        var response = await
            client.GetAsync("http://www.ietf.org/rfc/rfc2616.txt");
        response.EnsureSuccessStatusCode();
        var stream = await response.Content.ReadAsStreamAsync();
        var buffer = new byte[2*1024];
        var len = await stream.ReadAsync(buffer, 0, buffer.Length);
        var s = Encoding.ASCII.GetString(buffer, 0, len);
        Assert.True(s.Contains("Hypertext Transfer Protocol -- HTTP/1.1"));
    }
}
```

最后两种方法——ReadAsStringAsync 和 ReadAsByteArrayAsync——异步地提供消息内容的缓冲副本。ReadAsByteArrayAsync 返回原始的字节内容，而 ReadAsStringAsync 将内容解码为字符串返回。

```
[Fact]
public async Task HttpContent_can_be_consumed_as_a_string()
{
    using (var client = new HttpClient())
    {
        var response = await
            client.GetAsync("http://www.ietf.org/rfc/rfc2616.txt");
        response.EnsureSuccessStatusCode();
        var s = await response.Content.ReadAsStringAsync();
        Assert.True(s.Contains("Hypertext Transfer Protocol -- HTTP/1.1"));
    }
}
```

除了 HttpContent 实例的方法，静态类 HttpContentExtensions 也定义了扩展方法。所有扩展方法都是以下方法的变种：

```
public static Task<T> ReadAsAsync<T>(
    this HttpContent content,
    IEnumerable<MediaTypeFormatter> formatters,
    IFormatterLogger formatterLogger)
```

这个方法接受一列媒体类型格式化程序，尝试使用其中一个格式化程序，将消息内容读取为类型 T 的实例：

```
class GitHubUser
{
    public string login { get; set; }
    public int id { get; set; }
    public string url { get; set; }
    public string type { get; set; }
}

[Fact]
public async Task HttpContent_can_be_consumed_using_formatters()
{
```

```
        using (var client = new HttpClient())
        {
            var response = await
                client.GetAsync("https://api.github.com/users/webapibook");
            response.EnsureSuccessStatusCode();
            var user = await response.Content
                .ReadAsAsync<GitHubUser>(new MediaTypeFormatter[]
                {
                    new JsonMediaTypeFormatter()
                });
            Assert.Equal("webapibook", user.login);
            Assert.Equal("Organization", user.type);
        }
    }
```

媒体类型格式化程序（第 13 章将详细介绍）是抽象类 MediaTypeFormatter 的派生类，在对象和字节流表示（由因特网媒体类型定义）之间进行双向转换。

ReadAsAsync 方法也有一个过载方法不需要媒体类型格式化程序序列参数，而是使用一组默认的格式化程序。目前，这组默认的格式化程序是：JsonMediaTypeFormatter、XmlMediaTypeFormatter 和 FormUrlEncodedMediaTypeFormatter。

10.3.2　创建消息内容

在创建有效载荷不为空的消息时，我们按照内容的类型选择 HttpContent 的一个派生类实例，赋给 Content 类型属性。图 10-4 展示了一些可用的类。例如，如果消息内容是纯文本，那么我们可以用 StringContent 类来表示。

```
    [Fact]
    public void StringContent_can_be_used_to_represent_plain_text()
    {
        var response = new HttpResponseMessage()
            {
                Content = new StringContent("this is a plain text representation")
            };
        Assert.Equal("text/plain", response.Content.Headers.ContentType.MediaType);
    }
```

在默认情况下，Content-Type 标头设置为 text/plain，但这个值是可以修改的。

FormUrlEncodedContent 类用于生成名字 / 值对，按照 application/x-www-form-urlencoded 规则（HTML 表单也使用同样的编码规则）进行编码。这些名字 / 值对通过 FormUrlEncodedContent 构造函数的 IEnumerable<KeyValuePair<string,string>> 参数定义。

```
    [Fact]
    public async Task FormUrlEncodedContent_can_represent_name_value_pairs()
    {
        var request = new HttpRequestMessage
```

```
        {
            Content = new FormUrlEncodedContent(
                new Dictionary<string, string>()
                    {
                        {"name1", "value1"},
                        {"name2", "value2"}
                    })
        };
    Assert.Equal("application/x-www-form-urlencoded",
        request.Content.Headers.ContentType.MediaType);
    var stringContent = await request.Content.ReadAsStringAsync();
    Assert.Equal("name1=value1&name2=value2", stringContent);
}
```

编程模型还提供了另外三个类，在已经得到字节序列的内容时使用。如果内容包含在一个字节数组中，我们可以使用 ByteArrayContent 类。

```
[Fact]
public async Task ByteArrayContent_can_represent_byte_sequences()
{
    var alreadyExistantArray = new byte[] { 0x48, 0x65, 0x6c, 0x6c, 0x6f};
    var content = new ByteArrayContent(alreadyExistantArray);
    content.Headers.ContentType = new MediaTypeHeaderValue("text/plain")
        { CharSet = "utf-8" };
    var readText = await content.ReadAsStringAsync();
    Assert.Equal("Hello", readText);
}
```

StreamContent 和 PushStreamContent 类都用于流的处理：如果内容已经在流中（例如，从文件中读取得到），可以使用 StreamContent；而当内容由流输出产生时，应当使用 PushStreamContent。

StreamContent 实例创建时，构造函数中传入一个流。之后，在序列化 HTTP 消息时，HTTP 模型运行时会从这个流中拉取字节序列，添加到序列化的消息正文中。

```
[Fact]
public async Task StreamContent_can_be_used_when_content_is_in_a_stream()
{
    const string thisFileName = @"..\..\HttpContentFacts.cs";
    var stream = new FileStream(thisFileName, FileMode.Open, FileAccess.Read);
    using (var content = new StreamContent(stream))
    {
        content.Headers.ContentType = new MediaTypeHeaderValue("text/plain");

        // 断言
        var text = await content.ReadAsStringAsync();
        Assert.True(text.Contains("this string"));
    }
    Assert.Throws<ObjectDisposedException>(() => stream.Read(new byte[1], 0, 1));
}
```

当包含这个流的 StreamContent 实例释放时（例如，由 Web API 运行时释放），这个流会随之释放。

然而，在有的情况下，内容并没有准备好，而是由一个进程产生内容，这个进程需要使用一个流，将内容写入其中。一个典型的例子是使用 XmlWriter 进行 XML 序列化，需要向一个输出流中写入序列化的字节。一个解决办法是使用一个作为中介的 MemoryStream，向其中写入内容，然后将这个内存流传递给一个 StreamContent 实例。但是，这种办法会产生一个中间的副本，不太适合流媒体的使用场景。

一个更好的办法是使用 PushStreamContent 类。PushStreamContent 接受一个 Action<Stream, ...> 操作，以*推送方式*（push-style）运行：当运行时将流准备好时（例如，底层的 ASP.NET 响应内容流），就会调用 PushStreamContent 的流操作，该操作负责将内容写到最终的流中，不需要进行任何中间的缓冲。

```
[Fact]
public async Task
    PushStreamContent_can_be_used_when_content_is_provided_by_a_stream_writer()
{
    var xml = new XElement("root",
                            new XElement("child1", "text"),
                            new XElement("child2", "text")
        );
    var content = new PushStreamContent((stream, cont, ctx) =>
        {
            using (var writer = XmlWriter.Create(stream,
                new XmlWriterSettings { CloseOutput = true }))
            {
                xml.WriteTo(writer);
            }
        });
    content.Headers.ContentType =
        new MediaTypeWithQualityHeaderValue("application/xml");

    // 断言
    var text = await content.ReadAsStringAsync();
    Assert.True(text.Contains("<child1"));
}
```

要注意的重要一点是，这个操作不必同步地写入全部内容。实际上，只有当流关闭时，运行时才认为内容全部写入完毕，而不是在操作返回时完成。这意味着，可以在操作返回之后，由操作安排的代码将内容写入流（例如：异步任务或者定时器回调）。唯一的要求是，必须调用流的 Close 方法，以通知运行时内容已经完全写入。遗憾的是，如果一个错误在操作返回之后发生，那么这个错误的发生就没有办法通知给运行时。唯一可能的行为是关闭流，但是关闭流无法区分写入成功还是失败。为了解决这个问题，新版本的 System.Net.Http.Formatting.dll 程序集提供 PushStreamContent 的一个新的过载方法，接受一个 Func<Stream, HttpContent, TransportContext, Task>。使用这个新的过载方法，异步代码

可以返回一个 Task，用于通知运行时异常的发生。请注意，在下面一个示例中，Lambda 表示式使用了前缀 async，表明将返回一个 Task。

```
[Fact]
public async Task PushStreamContent_can_be_used_asynchronously()
{
    const string text = "will wait for 2 seconds without blocking";
    var content = new PushStreamContent(async (stream, cont, ctx) =>
    {
        await Task.Delay(2000);
        var bytes = Encoding.UTF8.GetBytes(text);
        stream.Write(bytes, 0, bytes.Length);
        stream.Close();
    });
    content.Headers.ContentType =
        new MediaTypeWithQualityHeaderValue("text/plain");

    // 断言
    var sw = new Stopwatch();
    sw.Start();
    var receivedText = await content.ReadAsStringAsync();
    sw.Stop();
    Assert.Equal(text, receivedText);
    Assert.True(sw.ElapsedMilliseconds > 1500);
}
```

前面介绍的代表消息内容的类都要求内容是字节序列。但是，新的 HTTP 编程模型也包含 ObjectContent 和 ObjectContent<T> 类，可以直接从对象定义 HTTP 消息内容。在其内部，这些类使用媒体类型格式化程序将对象转换为字节序列。

在接下来的示例中，我们为一个带有三个域的匿名类生成一个 JSON 表示。请注意，要使用的媒体类型格式化程序——在这个例子中为 JsonMediaTypeFormatter——必须明确地在 ObjectConent 构造函数中进行定义。

```
[Fact]
public async Task ObjectContent_uses_mediatypeformatter_to_produce_the_content()
{
    var representation = new
        {
         field1 = "a string",
         field2 = 42,
         field3 = true
        };
    var content = new ObjectContent(
        representation.GetType(),
        representation,
        new JsonMediaTypeFormatter());

    // 断言
    Assert.Equal("application/json",content.Headers.ContentType.MediaType);
    var text = await content.ReadAsStringAsync();
```

```
        var obj = JObject.Parse(text);
        Assert.Equal("a string", obj["field1"]);
        Assert.Equal(42, obj["field2"]);
        Assert.Equal(true, obj["field3"]);
    }
```

ObjectConent 既需要输入对象的值，也需要对象类型。其泛型版本，ObjectContent<T>，只是对 ObjectContent 进行了简化，将输入类型定义为泛型参数。

新编程模型的 HttpRequestMessageExtensions 类提供一组扩展方法，可以简化从请求创建响应的工作。例如，你可以创建一个响应消息，自动链接到对应的请求消息。

```
    [Fact]
    public void HttpRequestMessage_has_a_CreateResponse_extension_method()
    {
        var request =
            new HttpRequestMessage(HttpMethod.Get,
                new Uri("http://www.example.net"));
        var response = request.CreateResponse(HttpStatusCode.OK);
        Assert.Equal(request, response.RequestMessage);
    }
```

你可以使用 CreateResponse 的过载方法，指定媒体类型格式化程序，从对象创建一个表示，完成与 ObjectContent 类似的功能：

```
    public void CreateResponse_can_receive_a_formatter()
    {
        var request =
            new HttpRequestMessage(HttpMethod.Get,
                new Uri("http://www.example.net"));

        var response = request.CreateResponse(
            HttpStatusCode.OK,
            new { String = "hello", AnInt = 42 },
            new JsonMediaTypeFormatter());

        Assert.Equal("application/json",
            response.Content.Headers.ContentType.MediaType);
    }
```

在服务器驱动的内容协商场景中，需要使用请求信息——即请求的 Accept 标头——决定最合适的媒体类型，此时 CreateResponse 方法特别适合。

```
    [Fact]
    public void CreateResponse_performs_content_negotiation()
    {
        var request =
            new HttpRequestMessage(HttpMethod.Get,
                new Uri("http://www.example.net"));
        request.Headers.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/xml", 0.9));
```

```
        request.Headers.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json", 1.0));
        var response = request.CreateResponse(
            HttpStatusCode.OK,
            "resource representation",
            new HttpConfiguration());

        Assert.Equal("application/json",
            response.Content.Headers.ContentType.MediaType);
    }
```

请注意，这里使用的 CreateResponse 过载使用一个带有已配置格式化程序的 HttpConfiguration。

最后，你也可以选择创建定制的 HttpContent 派生类，生成消息内容。但是，在我们介绍这项技术之前，需要先理解 HTTP 消息内容的长度是如何计算的。

1. 内容长度和流

在 HTTP 中，定义有效载荷的正文长度主要有三种方法：

- 添加一个 Content-Length 标头字段，显式地指定消息长度；
- 使用分块传输编码（chunked transfer encoding），隐式地指定消息长度；
- 在所有内容传输完毕后关闭链接（只适用于响应内容），隐式地指定消息长度。

最后一种方法主要是为了与 HTTP 1.0 兼容，不应该使用，因为连接的异常中断会导致无法检测的内容损坏。

使用分块传输编码，消息正文分为一系列块（chunk），每块带有自己的大小定义。对于流媒体内容，其长度信息无法预知，如果使用分块传输编码，不需要缓冲就可以进行传输。

第一种方法比较简单，但是需要预先知道内容的长度。为此，HttpContent 类定义了如下的抽象方法：

```
protected internal abstract bool TryComputeLength(out long length)
```

每个具体的内容类，都必须按照其内容的表示方式，实现这个方法。例如：ByteArrayContent 类实现的 TryComputeLength 方法总是返回 true，提供底层的数组长度。而 PushStreamContent 类的实现返回 false，因为其内容是由注册的操作动态推送的。请注意，PushStreamContent 类无法知道这个操作将会推送多少个字节。最后，StreamContent 类的实现将这个查询委托给其构造函数中定义的底层 Stream：如果这个流是可定位的（seekable），那么 TryComputeLength 方法就使用 Stream.Length 计算内容长度；否则，TryComputeLength 方法会返回 false。

TryComputeLength 方法与 long? HttpContentHeaders.ContentLength 属性之间的关系也很紧密：如果这个属性值没有明确赋值，那么就会查询 TryComputeLength 方法。也就是说，我

们不需要明确指定 Content-Length 标头的值，除非需要从外部方法获得内容长度信息。还要注意，HttpContentHeaders.ContentLength 的类型是 long?，允许使用空值。

在第 11 章中，我们将会介绍托管层如何使用消息的内容长度，决定处理响应消息内容的最佳方式（即决定是否应该使用缓冲）。

2. 定制内容类

现在我们已经知道了消息内容的长度是如何定义的，以及什么影响流，接下来要介绍定制内容类的创建。接下来的一段代码定义了一个 FileContent 类，这个类派生自 HttpContent，是一个代表文件内容的定制类。

```
public class FileContent : HttpContent
{
    private readonly Stream _stream;
    public FileContent(string path,
        string mediaType = "application/octet-stream")
    {
        _stream = new FileStream(path, FileMode.Open, FileAccess.Read);
        base.Headers.ContentType = new MediaTypeHeaderValue(mediaType);
    }
    protected override Task
        SerializeToStreamAsync(Stream stream, TransportContext context)
    {
        return _stream.CopyToAsync(stream);
    }
    protected override bool TryComputeLength(out long length)
    {
        if (!_stream.CanSeek)
        {
            length = 0;
            return false;
        }
        else
        {
            length = _stream.Length; return true;
        }
    }
    protected override void Dispose(bool disposing)
    {
        _stream.Dispose();
    }
}
```

创建定制的 HttpContent 类，需要定义下面两个抽象方法：

```
protected internal abstract bool TryComputeLength(out long length);
protected abstract Task SerializeToStreamAsync(
    Stream stream, TransportContext context);
```

上一节曾经介绍过，第一个方法——TryComputeLength——用于尝试获得内容长度。在

FileContent 的实现中，这个方法使用 Stream.CanSeek 属性，查询是否能够计算文件流的长度，如果文件流长度可以计算，就使用 Stream.Length 属性，返回内容长度。

第二个方法，SerializeToStreamAsync，负责将内容写到传入的 Stream。这个方法可以异步操作，在写入完成前返回一个 Task。当写入过程结束时，这个返回的 Task 应该得到通知。当消息内容由另一个异步过程提供时（例如：从文件系统或外部系统读入），SerializeToStreamAsync 方法的这种异步能力就非常有用。例如，FileContent 的实现使用了 .NET 4.5 中引入的 CopyToAsync 方法，开始异步复制操作，并返回代表这一操作的 Task。

除了直接继承 HttpContent 类，你也可以采取另一种方法：使用 StreamContent 和 PushStreamContent 类。你可以继承 StreamContent 或 PushStreamContent 类，或者通过工厂方法使用这两个类。下面的代码创建了 PushStreamContent 的一个派生类，不需要使用任何缓冲，就可以构建基于 XML 的内容。

```
public class XmlContent : PushStreamContent
{
    public XmlContent(XElement xe)
        : base(PushStream(xe), "application/xml")
    {
    }

    private static Action<Stream,HttpContent,TransportContext>
        PushStream(XElement xe)
    {
        return (stream, content, ctx) =>
            {
                using (var writer = XmlWriter.Create(stream,
                    new XmlWriterSettings(){CloseOutput = true}))
                {
                    xe.WriteTo(writer);
                }
            };
    }
}
```

因为 PushStreamContent 的构造函数需要一个 Action<Stream,HttpContent, TransportContext>，前面的示例中使用了一个私有的静态方法，从提供的 XElement 创建了这个操作。还需要注意的是，代码中使用了 XmlWriterSettings 参数，以关闭提供的流。因为操作假定是异步的，关闭流就标识着这一过程的终结。

我们可以对 XElement 使用一个扩展方法，实现同样的功能：

```
public static class XElementContentExtensions
{
    public static HttpContent ToHttpContent(this XElement xe)
    {
        return new PushStreamContent((stream, content, ctx) =>
            {
```

```
                using (var writer = XmlWriter.Create(stream,
                    new XmlWriterSettings(){CloseOutput = true}))
                {
                    xe.WriteTo(writer);
                }
            },"application/xml");
        }
    }
```

10.4 小结

本章，我们关注的是新的 HTTP 编程模型，这一编程模型在 .NET 框架的 4.5 版本中引入，是 Web API 以及新 `HttpClient` 类的核心。正如我们介绍的，这个新的编程模型提供了实现可用性和可测试性的更好方式，用于处理 HTTP 的核心概念——消息、标头和内容。之后的章节将在此基础之上，帮助你更深入理解 Web API 的内部工作方式。也就是说，第 11 章将介绍 HTTP 编程模型与一个底层 HTTP 栈（例如 ASP.NET 提供的 HTTP 栈）之间的接口。

第 11 章

托管

Web API 遇见了楼下的邻居。

第 4 章将 ASP.NET Web API 处理架构分为三层：托管、消息处理程序管道和控制器处理。本章要详细介绍其中的第一层：托管。

托管层实际上是一个*宿主适配层*（host adaptation layer），这一层作为一个桥梁，连接 Web API 处理架构和所支持的外部托管基础结构。实际上，Web API 自身并没有托管机制。恰恰相反，Web API 的目标是独立于宿主，可以在多种托管场景下使用。

概括起来，宿主适配层负责执行如下任务：

- 创建和初始化消息处理器管道，将其封装在 `HttpServer` 实例中；
- 从底层的托管基础结构接收 HTTP 请求。这一任务通常通过注册一个回调函数实现；
- 将 HTTP 请求从本地表示(如 ASP.NET 的 `HttpRequest`)转换成 `HttpRequestMessage` 实例；
- 将这些 `HttpRequestMessage` 实例推送到消息处理器管道，以此开始 Web API 请求处理；
- 当一个响应生成并返回时，宿主适配层将返回的 `HttpResponseMessage` 实例转换成底层基础结构的本地响应表示（如 ASP.NET 的 `HttpResponse`），传递给底层的托管基础结构。

ASP.NET Web API 1.0 支持两种托管适配层：*Web 托管*和*自托管*。使用 Web 托管适配层，我们可以在经典的 IIS 支持的 ASP.NET 托管基础结构上使用 Web API。使用自托管适配层，我们可以在任何 Windows 进程（即：控制台应用程序和 Windows 服务）上使用 Web API。这两种托管适配层可以通过独立的 NuGet 软件包获得：`Microsoft.AspNet.WebApi.WebHost` 和 `Microsoft.AspNet.WebApi.SelfHost`。ASP.NET Web API 2.0 引入了 OWIN 托管适配层，可以通过 `Microsoft.AspNet.WebApi.Owin` 软件包下载。使用这个新的托管适配层，

我们可以在任何 OWIN 兼容的宿主上运行 Web API。

本章的目的是为你提供必要的知识，在这些托管场景中充分使用 Web API。为了实现这一目标，我们将更为深入地介绍这些托管适配层，关注其内部行为。我们还会以 Azure Service Bus 适配层为例，通过 Service Bus 中继，将私有托管的 Web API 应用安全提供给公共 Web，详细介绍如何支持新的托管场景。

最后，我们还将介绍一个针对测试场景，通常称为*内存托管*（in-memory hosting）的特殊托管方式。这种托管方式直接将一个 `HttpClient` 实例连接到一个 Web API `HttpServer` 实例，客户端和服务器可以进行在内存中直接进行 HTTP 通信。

掌握了这些内部的实施架构，我们可以正确地进行 Web API 托管的配置和优化（如消息缓冲）。如果你想编写一个定制的宿主或者扩展现有的宿主，本章的内容也会有所帮助。

Web API 的托管机制涉及若干外部技术，例如：经典的 ASP.NET 管道、WCF 信道栈层（channel stack layer）或 OWIN 规范。为了内容的完整性，本章也会简要介绍这些外部技术，尤其是其中与托管相关的部分。

11.1　Web托管

所谓的 *Web 托管*使用经典的 ASP.NET 管道。接下来，我们将首先回顾 ASP.NET 管道的相关内容，然后简要描述 Web API 和 ASP.NET MVC 使用的 ASP.NET 路由基础结构，最后介绍 Web API 如何将二者集成在一起。

11.1.1　ASP.NET基础结构

如图 11-1 所示，ASP.NET 基础结构有三个主要元素：应用程序、模块和处理程序。

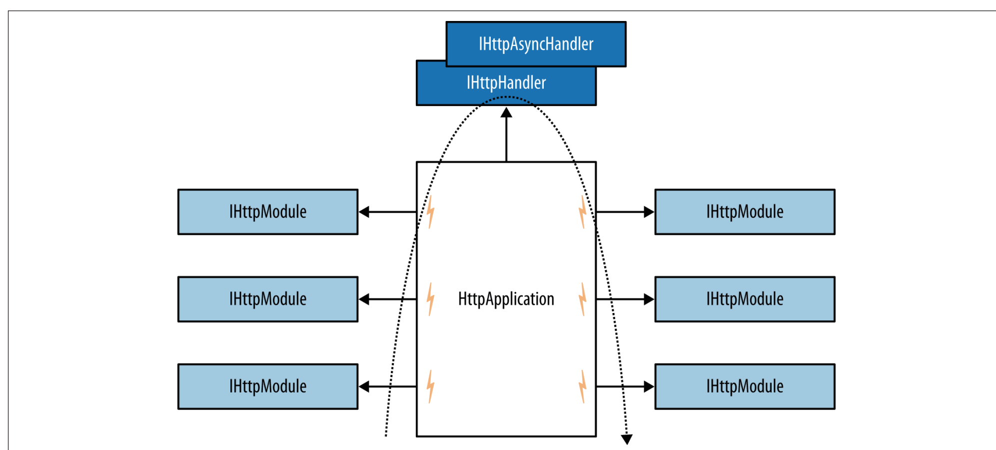


图 11-1：ASP.NET 管道

1. 应用程序

在 ASP.NET 中，部署单元是应用程序，表示为 HttpApplication 类。通过定义一个定制的 global.asax 文件，你可以为每个应用程序创建一个特殊的派生类。

如果一个请求映射到一个应用程序，运行时就会创建或者选择一个 HttpApplication 实例，处理这个请求。运行时还会创建一个上下文，表示 HTTP 请求和响应：

```
public sealed class HttpContext : IServiceProvider
{
    public HttpRequest Request { get {...} }
    public HttpResponse Response { get {...} }
    ...
}
```

应用程序接着会让这个上下文“流经”一组管道阶段，这些阶段由 HttpApplication 成员事件表示。例如：请求开始处理时会触发 HttpApplication.BeginRequest 事件。

2. 模块

一个应用程序包含一组注册的模块类，实现 IHttpModule 接口：

```
public interface IHttpModule
{
    void Dispose();
    void Init(HttpApplication context);
}
```

一个新的应用程序对象在构建时，会为每个模块类创建一个实例，并调用这些实例的 IHttpModule.Init 方法。每个模块利用这个调用，将自己附加（attach）在想要处理的管道事件上。一个模块可以附加在多个事件上，一个事件可以有多个模块附加其上。

随后这些模块可以作为过滤程序，在 HTTP 请求和响应流经事件管道时对其进行处理。这些模块也可以提前终止请求处理，立刻产生响应。

3. 处理程序

在触发所有的请求事件之后，应用程序选择一个处理程序（表示为 IHttpHandler 或 IHttpAsyncHandler 接口），将请求的处理工作委托给这个处理程序：

```
public interface IHttpHandler
{
    void ProcessRequest(HttpContext context);
    bool IsReusable { get; }
}
public interface IHttpAsyncHandler : IHttpHandler
{
    IAsyncResult BeginProcessRequest(HttpContext context,
        AsyncCallback cb, object extraData);
    void EndProcessRequest(IAsyncResult result);
}
```

当处理程序的处理结束时，这个上下文流回应用程序管道，触发响应事件。

处理程序是应用程序最终委托请求处理的端点（endpoint），构成了基于 ASP.NET 基础结构的多个框架（如 Web Forms、ASP.NET MVC 或 Web API）使用的主要集成点。

例如，在 ASP.NET Web Forms 框架中，`System.Web.UI.Page` 类实现了 `IHttpHanlder` 接口。也就是说，类与一个 .aspx 文件结合构成了一个处理程序，如果请求 URI 与这个 .aspx 文件路径匹配，应用程序就会调用这个处理程序。

我们通过将请求 URI 映射到应用程序目录中的一个文件（例如：一个 .aspx 文件），或者使用 ASP.NET 路由功能，来选择处理程序。ASP.NET Web Forms 使用前一种技术[1]，而 ASP.NET MVC 使用后一种。Web API 也使用 ASP.NET 路由功能，接下来将进行介绍。

11.1.2 ASP.NET路由

在 ASP.NET 基础结构中，配置路由的通常方法是将路由信息添加到 `RouteTable.Routes` 静态属性中，这个属性是一个 `RouteCollection`。

例如，示例 11-1 展示了 ASP.NET MVC 项目模板定义的默认映射，这些信息通常定义在 global.asax 文件中。

示例 11-1：ASP.NET MVC 默认路由配置

```
protected void Application_Start()
{
    ...
    RegisterRoutes(RouteTable.Routes);
    ...
}

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        "Default", // 路由名
        "{controller}/{action}/{id}", // 带参数的 URL
        new { controller = "Home", action = "Index",
                                id = UrlParameter.Optional }
    );
}
```

静态属性 `RouteTable.Routes` 定义了一个路由集合，对应用程序全局有效，具体的路由就添加在这个路由集合中。示例 11-1 中使用了 `MapRoute` 方法添加路由，这个方法并不是路由集合的实例方法，而是 ASP.NET MVC 中引入的一个扩展方法，用于添加 MVC 相关的

注 1：Web Forms 也可以通过 `RouteCollection.MapPageRoute` 方法，使用路由功能。

路由。我们随后将看到，Web API 也使用类似的方式。

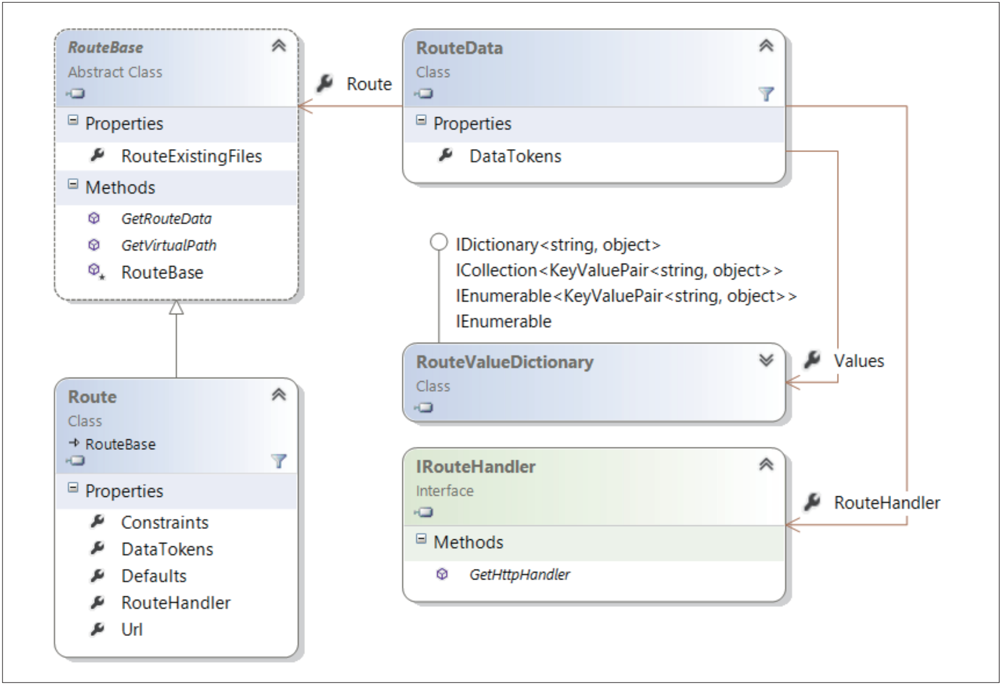


图 11-2：ASP.NET 路由类

图 11-2 展示了参与 ASP.NET 路由过程的一些类。通用的路由概念定义为抽象类 `RouteBase`，类中包含 `GetRouteData` 实例方法。`GetRouteData` 检查一个请求上下文（即请求 URI）是否与路由匹配，如果匹配，就返回包含一个 `RouteData` 实例，其中包含一个简单的处理程序工厂 `IRouteHandler`。此外，`RouteData` 还包含由匹配过程产生的一组额外的值。例如：示例 11-1 中的 `Default` 路由所匹配的 HTTP 请求，产生的 `RouteData` 会包含 `controller` 和 `action` 路由值。

`RouteBase` 类还包含 `GetVirtualPath` 方法，用于执行反向查找：给出一组值，返回一个匹配该路由并能产生这些值的 URI。

抽象类 `RouteBase` 并不与具体的路由匹配过程相关联，而是由具体的派生类进行定制。`Route` 是 `RouteBase` 的派生类之一，定义了一个具体的匹配过程，该过程用到如下元素：

- 一个 URI 模板，定义 URI 结构（例如：`"{controller}/{action}/{id}"`）；
- 一组默认值（例如：`new { controller = "Home", action = "Index", id = UrlParameter.Optional }`）；
- 一组附加约束。

URI 模板既定义了匹配该路由的 URI 所必须具有的结构，也定义了占位符，用于从 URI 路径段中获取路由数据。

对于一个请求，附加在 `PostResolveRequestCache` 管道事件上的 `UrlRoutingModule` 执行路由选择逻辑。对每个请求，这个模块将当前的请求与全局的 `RouteTable.Routes` 集合中的路由进行匹配，如果匹配成功，相关的 HTTP 处理程序就会映射到当前的请求。因此，在管道末端，应用程序会将请求的处理委托给这个处理程序。例如，由 MVC 的 `MapRoute` 扩展方法添加的所有路由都映射到一个特殊的 `MvcHandler`。

11.1.3　Web API路由

ASP.NET 路由模型和基础结构依赖于遗留的 ASP.NET 模型，即：通过 `HttpContext` 示例来表示请求和响应。Web API 虽然使用类似的路由概念，但是也使用新的 HTTP 类模型，因此定义了一组新的路由相关类和接口，如图 11-3 所示。

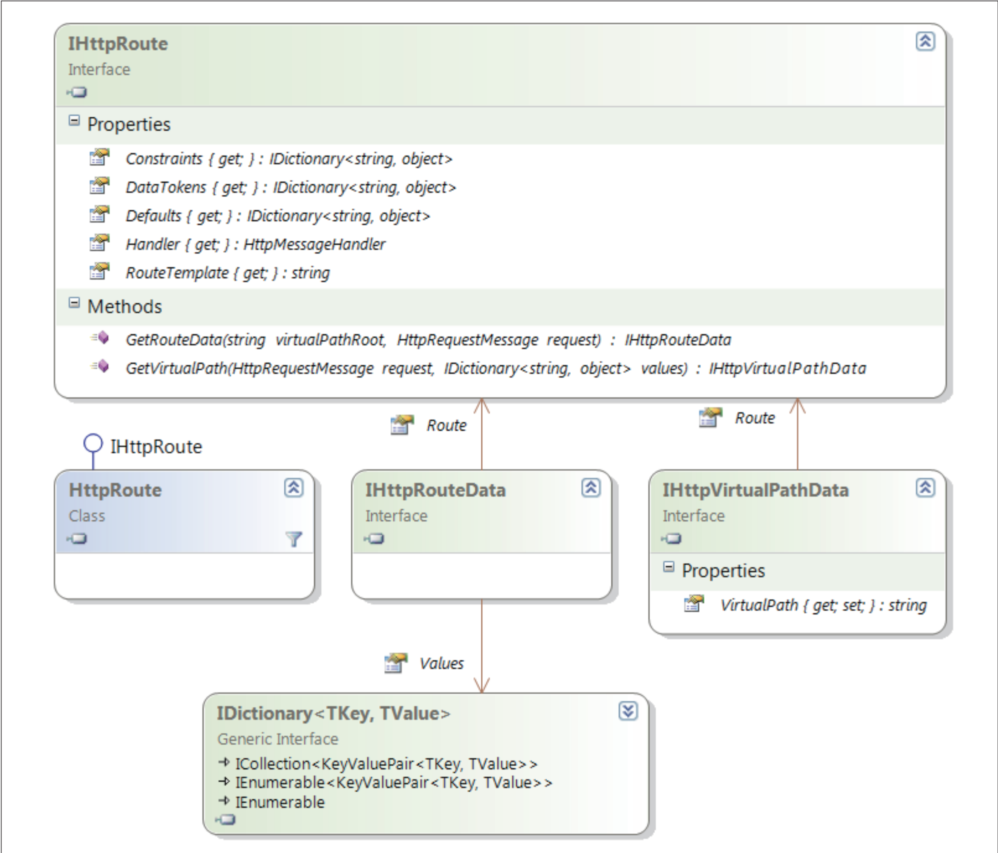


图 11-3：Web API 路由类

IHttpRoute 表示一个 Web API 路由，与经典的 ASP.NET Route 类具有类似特征，包括：

- GetRouteData 方法，接收一个 HTTP 请求和虚拟根路径，返回一个包含值字典的 IHttpRouteData；
- GetVirtualPath 方法，接收一个值字典和一个请求消息，返回一个带有 URI 的 IHttpVirualPath；
- 一组属性，包含路由模板、默认路由和约束。

Web API 的一个重要区别是，Web API 使用新的 HTTP 类模型——特别是 HttpRequestMessage 和 HttpMessageHanlder 类 —— 而不是旧的 ASP.NET 类，如 HttpRequest 和 IHttqHandler。

Web API 还定义了一种方法，在经典的 ASP.NET 路由基础结构之上使用新路由类（如图 11-4 所示）。当 Web API 托管在 ASP.NET 之上时，使用这个内部适配层，可以让我们在一个 HTTP 应用程序内同时使用新旧两种路由。

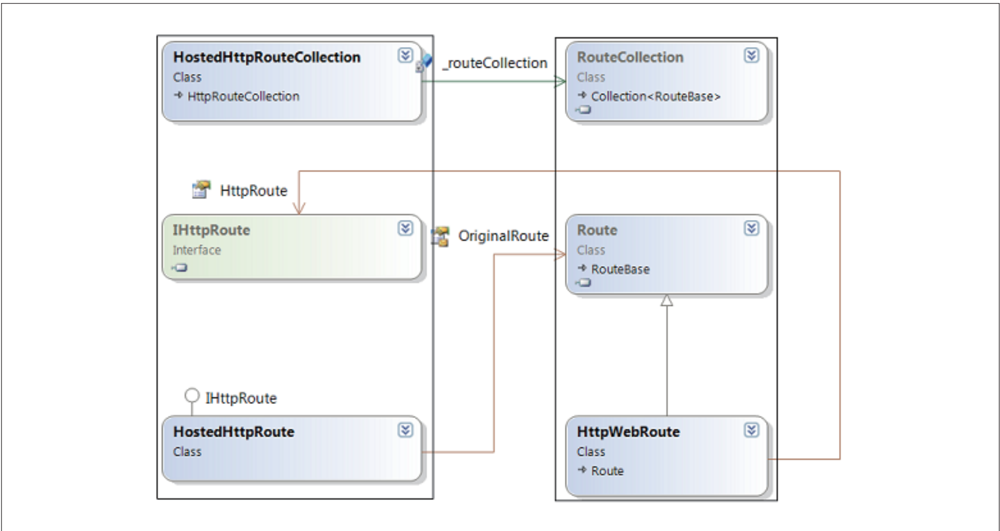


图 11-4：Web API 路由适配类

HostedHttpRouteCollection 类是一个适配器，在经典的 ASP.NET RouteCollection 上提供一个 ICollection<IHttpRoute> 接口。如果一个新的 IHttpRoute 加入这个集合，HostedHttpRouteCollection 将其包装成一个特殊的适配器 Route（HttpWebRoute），并将其添加到 ASP.NET 的路由集合中。

通过这种方式，这个全局的 ASP.NET 路由集合可以同时包含经典的路由，和新 Web API 路由的适配器。

11.1.4 全局配置

当托管在 ASP.NET 之上时，Web API 相关的配置定义在一个 `HttpConfiguration` 单例对象中，可以通过静态的 `GlobalConfiguration.Configuration` 属性访问。在示例 11-2 中，这个单例对象用做默认路由配置的参数。

示例 11-2：ASP.NET Web API 默认路由配置

```
// config 等同 GlobalConfiguration.Configuration
config.MapHttpAttributeRoutes();
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

这个单例配置的 `Routes` 属性指向一个 `HostedHttpRouteCollection`，其中封装了全局的 `RouteTable.Routes` 集合（如图 11-5 所示）。也就是说，所有添加到 `GlobalConfiguration.Configuration.Routes` 的 Web API 路由，最后都会作为经典的 ASP.NET 路由，添加到全局的 `RouteTable.Routes` 集合中。因此，当 `UrlRoutingModule` 尝试寻找一个匹配路由时，这些 Web API 路由也会纳入寻找范围。

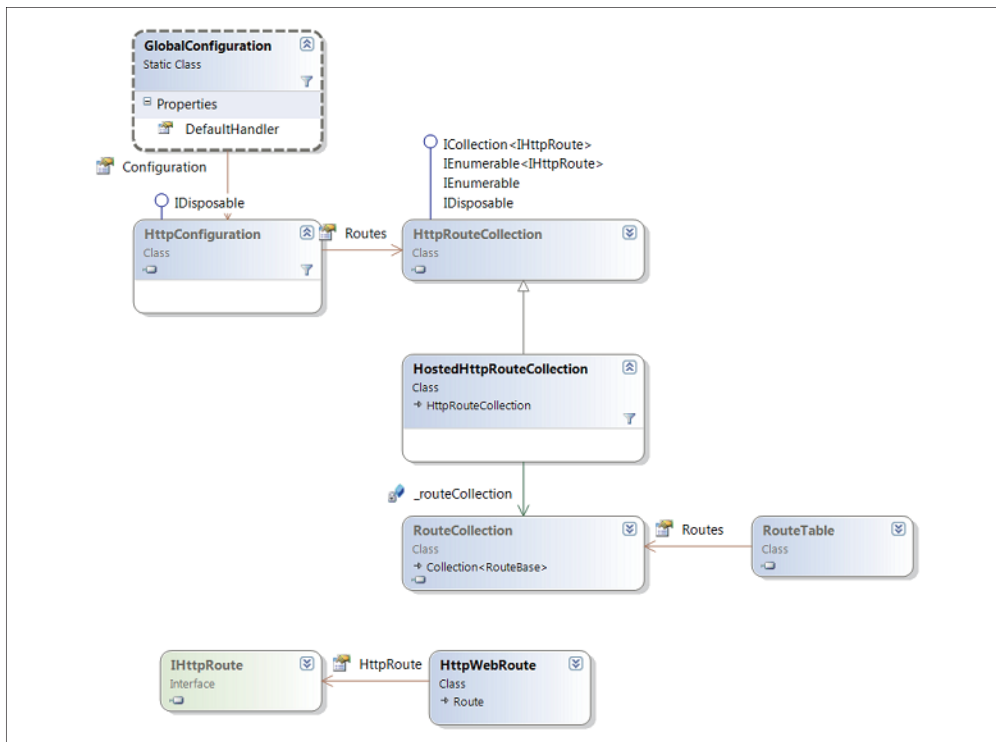


图 11-5：Web API 全局配置

由 ASP.NET MVC 配置添加的路由关联到 MvcHanlder 类。这意味着，在管道末端，匹配这些路由的所有请求都会委托给这个 MvcHandler 处理。然后，这个特殊的处理程序会执行 MVC 相关的请求处理，即：选择控制器并调用映射操作。

Web API 的场景非常类似：由 GlobalConfiguration.Configuration.Routes 添加的路由关联到 HttpControllerHandler，最终由这个 HttpControllerHandler 处理匹配这些 Web API 路由的所有请求。

图 11-6 展示了这一特点，RouteTable.Routes 集合既包含 MVC 路由，也包含 Web API 路由。但是，请注意 MVC 路由关联到 MvcHandler，而 Web API 路由关联到 HttpController Handler。

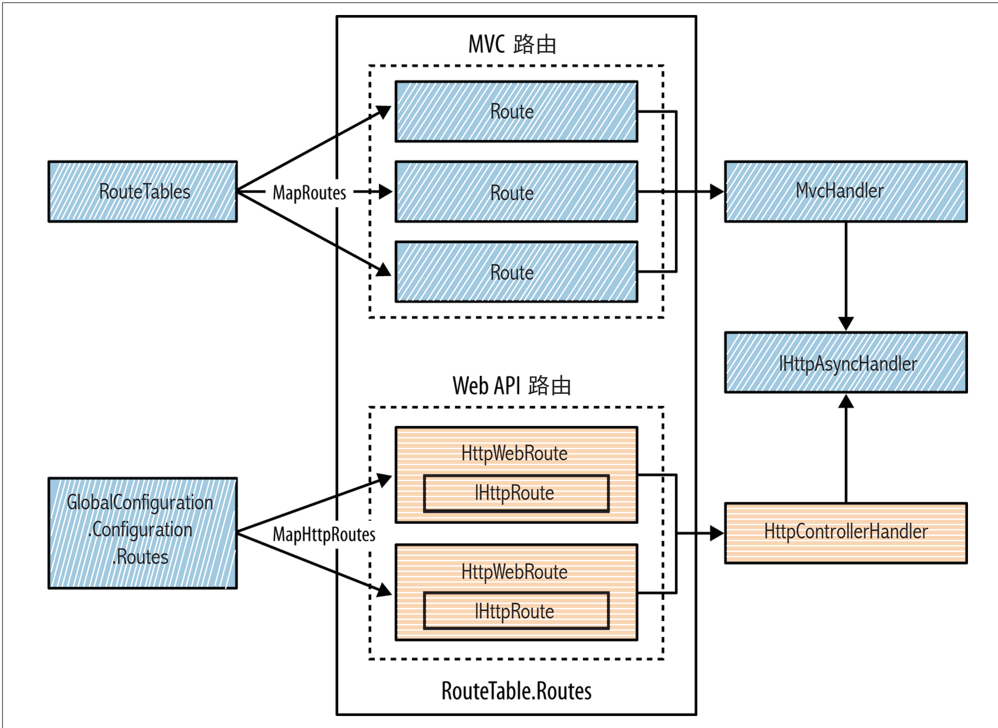


图 11-6：RouteTable.Routes 既包含 MVC 路由，也包含 Web API 路由

11.1.5 Web API ASP.NET处理程序

我们刚才介绍过，匹配 Web API 路由的所有 ASP.NET 请求都会由新的 HttpController Handler 进行处理。这个处理程序的 BeginProcessRequest 方法会执行以下操作。

- 首先，当处理第一个请求时，这个处理程序会通过 GlobalConfiguration.Configuration，延迟生成一个 HttpServer 单例。这个服务器实例包含了消息处理程序管道，其中包括控制器分发处理程序。

- 接下来，处理程序将当前 HttpContext 中的 ASP.NET HttpRequest 消息转换成一个新的 HttpRequestMessage 实例。
- 最后，处理程序将这个 HttpRequestMessage 推送到 HttpServer 实例，开始 Web API 处理中与平台无关的阶段，这一阶段由消息处理程序管道和控制器层组成。

本地 ASP.NET 消息表示和 Web API 表示之间的转换，是通过一个*服务对象*（service object）配置的，这个服务对象实现 IHostBufferPolicySelector 接口。IHostBufferPolicySelector 接口有两个方法：UseBufferedInputStream 和 UseBufferedOutputStream，定义消息正文是否应该进行缓冲。HttpControllerHandler 从全局配置请求得到这个服务对象，用于判断：

- HttpRequestMessage 内容使用的请求输入流是 ASP.NET 缓冲输入，还是流输入；
- HttpResponseMessage 内容是否写入一个 ASP.NET 缓冲输出流。

默认注册的 Web 宿主策略总是使用缓冲输入流。对于输出流，默认策略基于返回的 HttpResponseMessage 属性，使用如下规则。

- 如果内容长度已知，那么就明确设置 Content-Length，不使用分块。因为内容长度已经确定，无需缓冲即可进行传输。
- 如果内容类为 StreamContent，那么只有当底层流不提供长度信息时，才使用*分块*传输编码。
- 如果内容类为 PushStreamContent，那么就使用*分块*传输编码。
- 如果不属于上述三种情况，则在内容传输前进行缓冲，以判断其长度，不使用分块。

将 ASP.NET HttpRequest 消息转换成一个新的 HttpRequestMessage 实例的操作，不仅仅处理请求消息的信息，还获取一组托管上下文信息，插入 HttpRequestMessage.Properties 字典中。这一信息的内容有：

- 客户端使用的证书（如果请求是在使用客户端认证的 SSL/TLS 连接上完成的）；
- 一个布尔值属性，表明请求是否来自同一机器；
- 一个标识（如果开启了定制错误）。

请注意，这个信息并非来自请求消息，而是由托管基础结构提供，反映了托管上下文，如连接特征。例如：作为属性添加到消息的客户端认证信息，可以通过扩展方法 GetClientCertificate 公开访问。其余的信息专门供 Web API 运行时使用。

ASP.NET Web API 的 2.0 版本引入了*请求上下文*（request context）的概念，用于组织所有的上下文信息。新的 HttpRequestContext 类不再包含各种无类型的请求属性，而是用下面一组属性来代表上下文信息：

- 客户端证书；
- 虚拟根路径；

- 请求身份信息。

Web API ASP.NET 处理程序创建一个 `WebHostHttpRequestContext` 实例（`WebHostHttpRequestContext` 派生自 `HttpRequestContext`），在其中填入请求的上下文信息。其上层次就可以通过这个请求的 `GetRequestContext` 扩展方法，访问这一信息。

图 11-7 图形化地概括了 Web 托管的基础结构，展示了路由解析过程，以及分发至 `HttpServer` 实例的操作。总结如下。

- Web API 可以在 ASP.NET 基础结构之上进行托管，与其他框架（如 ASP.ENT MVC 或 Web Forms）共享同样的应用程序。
- ASP.NET 路由基础结构用于识别绑定到 Web API 运行时的请求。这些请求交由特殊的处理程序处理，从本地 HTTP 表示转换为新的 `System.Net.Http` 模型对象。
- 我们可以在应用程序开始时进行 Web API 配置，通常做法是在 global.asax.cs 文件的 `Application_Start` 方法中添加代码，使用 `GlobalConfiguration.HttpConfiguration` 单例对象进行配置。
- 使用 Web 托管方式时，我们必须在底层的宿主上，而非通用的 Web API 上进行一些配置定义。例如：我们可以使用 IIS 管理器，配置安全连接使用 SSL 或 TLS。

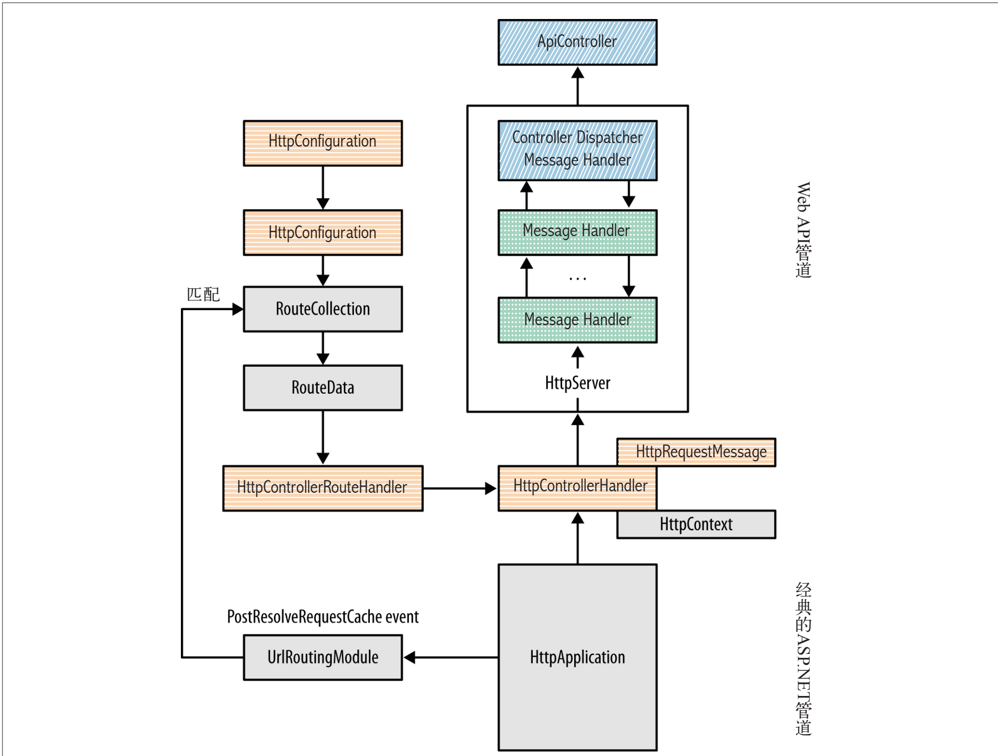


图 11-7：ASP.NET 托管架构

11.2 自托管

Web API 也包含自托管（即在任何 Windows 进程上托管，例如：控制台应用程序或者 Windows 服务）的适配层。示例 11-3 展示了自托管的典型代码。

示例 11-3：自托管

```
var config = new HttpSelfHostConfiguration("http://localhost:8080");
config.Routes.MapHttpRoute("default", "{controller}/{id}",
                                new { id = RouteParameter.Optional });
var server = new HttpSelfHostServer(config);
server.OpenAsync().Wait();
Console.WriteLine("Server is opened");
Console.ReadLine();
server.CloseAsync().Wait();
```

请注意，采用自托管方式时，我们必须明确地创建、配置并启动一个服务器实例。这与 Web 托管方式不同。采用 Web 托管时，所需的 `HttpServer` 实例是由 ASP.NET 处理程序隐式延迟创建的。还需要注意的是，示例 11-3 使用了自托管场景专用的类。`HttpSelfHostServer` 类派生自通用类 `HttpServer`，由一个 `HttpSelfHostConfiguration` 进行配置，这个 `HttpSelfHostConfiguration` 派生自通用类 `HttpConfiguration`。托管的基地址明确定义在自托管配置中。图 11-8 展示了这些类之间的关系。

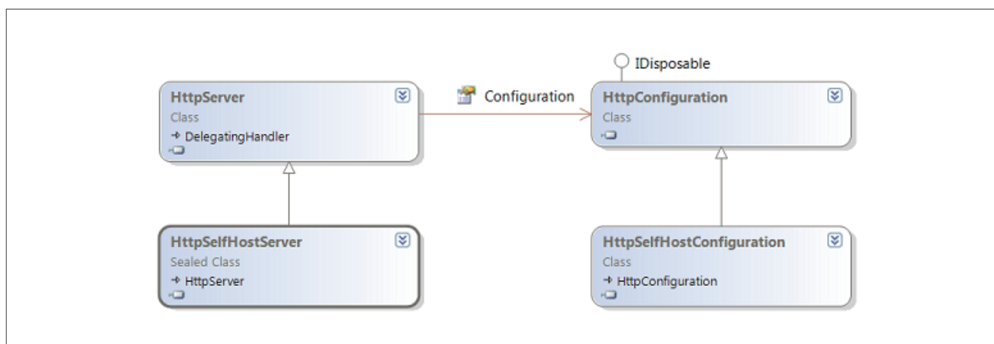


图 11-8：自托管服务器和配置类

在 Web API 的 1.0 版本中，`HttpSelfHostServer` 内部使用了 WCF（Windows Communication Foundation）信道栈层，从底层 HTTP 基础结构获得请求消息。下一节将简要介绍 WCF 的高层架构，以便在此基础上，进行 Web API 自托管特点的介绍。

11.2.1 WCF架构

WCF 架构分为两层：信道栈层和服务模型层（如图 11-9 所示）。

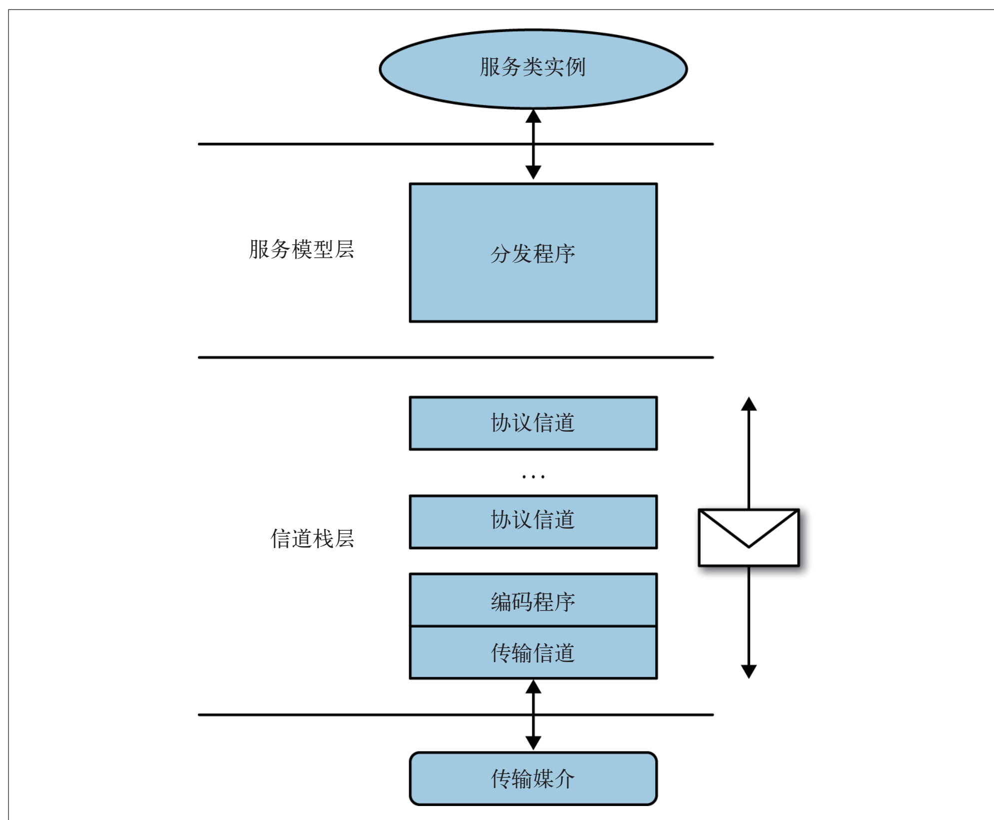


图 11-9：WCF 架构

底层的信道栈层由一个信道栈组成，与经典的网络协议栈行为方式类似。这些信道分为两类：传输信道和协议信道。协议信道处理在栈中向上和向下流动的消息。协议信道的典型用法有：在发送端添加数字签名，以及在接收端验证这些签名。传输信道处理与传输媒介的交互（例如：TCP、MSMQ 和 HTTP），即传输和发送消息。传输信道使用编码程序（encoder），在传输媒介字节流和消息实例之间进行转换。

上层的服务模型层执行消息和方法调用之间的交互，处理如下任务：

- 将收到的消息转换成为一个参数序列；
- 获得要使用的服务实例；
- 选择要调用的方法；
- 获得调用方法的线程。

信道栈层的具体组织由绑定进行描述，如图 11-10 所示。一个绑定是绑定元素（binding element）的一个有序集合，其中每个元素大致描述一个信道或编码程序。第一个绑定元素描述上层信道，最后一个元素描述底层信道，底层信道总是一个传输信道。

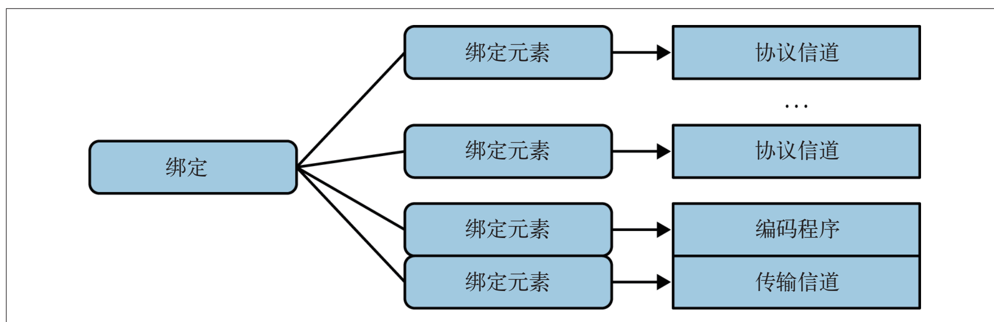


图 11-10：绑定、绑定元素以及信道

11.2.2　HttpSelfHostServer类

HttpSelfHostServer 类实现了一个自托管的 Web API 服务器。如示例 11-3 中的代码所示，这个自托管 Web API 服务器使用 HttpSelfHostConfiguration 类的一个实例进行配置，HttpSelfHostConfiguration 类派生自较为通用的 HttpConfiguration，加入了与自托管场景相关的特殊配置属性。

HttpSelfHostServer 内部创建了一个 WCF 信道栈，使用这个信道栈监听 HTTP 请求。Web API 自托管支持引入了新的 HttpBinding 类，描述这个信道栈。

当启动服务器时，HttpSelfHostServer.OpenAsync 方法创建一个 HttpBinding 实例，让 HttpSelfConfiguration 实例配置这个 HttpBinding 实例，然后使用这个绑定异步创建 WCF 信道栈。HttpSelfHostServer.OpenAsync 还创建一个“泵”，不断从这个信道栈中拉取消息，将消息转换为 HttpRequestMessage 实例，并将这些新请求推送到消息处理程序管道中。

与 Web 托管的情况类似，创建的 HttpRequestMessage 带有一个 HttpRequestContext 实例，其中包含了从托管上下文得到的一组属性。当客户端认证使用 TLS/SSL 时，这组属性包含客户端认证信息。

从信道栈拉取信息的泵还负责获得返回的 HttpResponseMessage，将其写入信道栈中。对于响应流，自托管与 Web 托管处理方式不同。自托管要么使用明确的 Content-Length 标头，要么使用分块传输编码，选择基于如下 HttpSelfHostConfiguration 选项：

```
public TransferMode TransferMode {get; set;}
```

如果选项值为 TransferMode.Buffered，那么不管 TryComputeLength 的返回值或 ContentLength 标头属性值是什么，宿主总是明确设置 Conent-Length。也就是说，如果 HttpContent 实例没有提供长度信息，那么宿主会在内存中缓冲全部内容，以判断其长度，然后才发送内容。另外，如果选项值为 TransferMode.Streamed，那么即便已知内容长度，宿主也总是使用分块传输。

11.2.3　HttpSelfHostConfiguration类

我们前面提到，HttpSelfHostServer 中定义的 HttpSelfHostConfiguration 负责配置内部使用的 HttpBinding，HttpBinding 又配置 WCF 消息信道。因此，HttpSelfHostConfiguration 类包含一组公共属性（参见示例 11-4），反映了这个内部实现的细节（即基于 WCF 编程模型）。例如：MaxReceivedMessageSize（在常用的 WCF BasicHttpBinding 类中也有这个属性）定义了接收到消息的最大长度。另一个例子是 X509CertificateValidator 属性，这个属性基于 System.IdentityModel 程序集中的一个类，用于配置 SSL/TLS 连接上收到的客户端认证。

示例 11-4：HttpSelfHostConfiguration 属性

```
public class HttpSelfHostConfiguration : HttpConfiguration
{
    public Uri BaseAddress {get;}
    public int MaxConcurrentRequests {get;set;}
    public TransferMode TransferMode {get;set;}
    public HostNameComparisonMode HostNameComparisonMode {get;set;}
    public int MaxBufferSize {get;set;}
    public long MaxReceivedMessageSize {get;set;}
    public TimeSpan ReceiveTimeout {get;set;}
    public TimeSpan SendTimeout {get;set}
    public UserNamePasswordValidator UserNamePasswordValidator {get;set;}
    public X509CertificateValidator X509CertificateValidator {get;set;}
    public HttpClientCredentialType ClientCredentialType {get;set;}

    // 为清晰起见，省略其他成员
}
```

进行内部自托管行为的配置的另一种方法是创建 HttpSelfHostConfiguration 的一个派生类，过载 OnConfigureBinding 方法。这个方法接收 HttpSelfHostServer 内部创建的 HttpBinding 实例，可以在其用于配置 WCF 信道栈之前，修改绑定设置。图 11-11 展示了自托管架构，特别是 WCF 信道栈的使用，以及配置和 WCF 绑定之间的关系。

Web API 自托管依赖于 WCF，既有好处，也有缺点。主要的好处是，我们可以使用 WCF HTTP 绑定的大部分功能，例如：消息限制（message limiting）、节流（throttling）和超时（timeout）。主要缺点是，对 WCF 的依赖是通过 HttpSelfHostConfiguration 公共接口，也就是这个接口的一些属性提供的。

请注意，消息泵从底层信道栈获取消息，并将其转换成为 HttpRequestMessage 实例，之后才将其推送给 HttpServer。

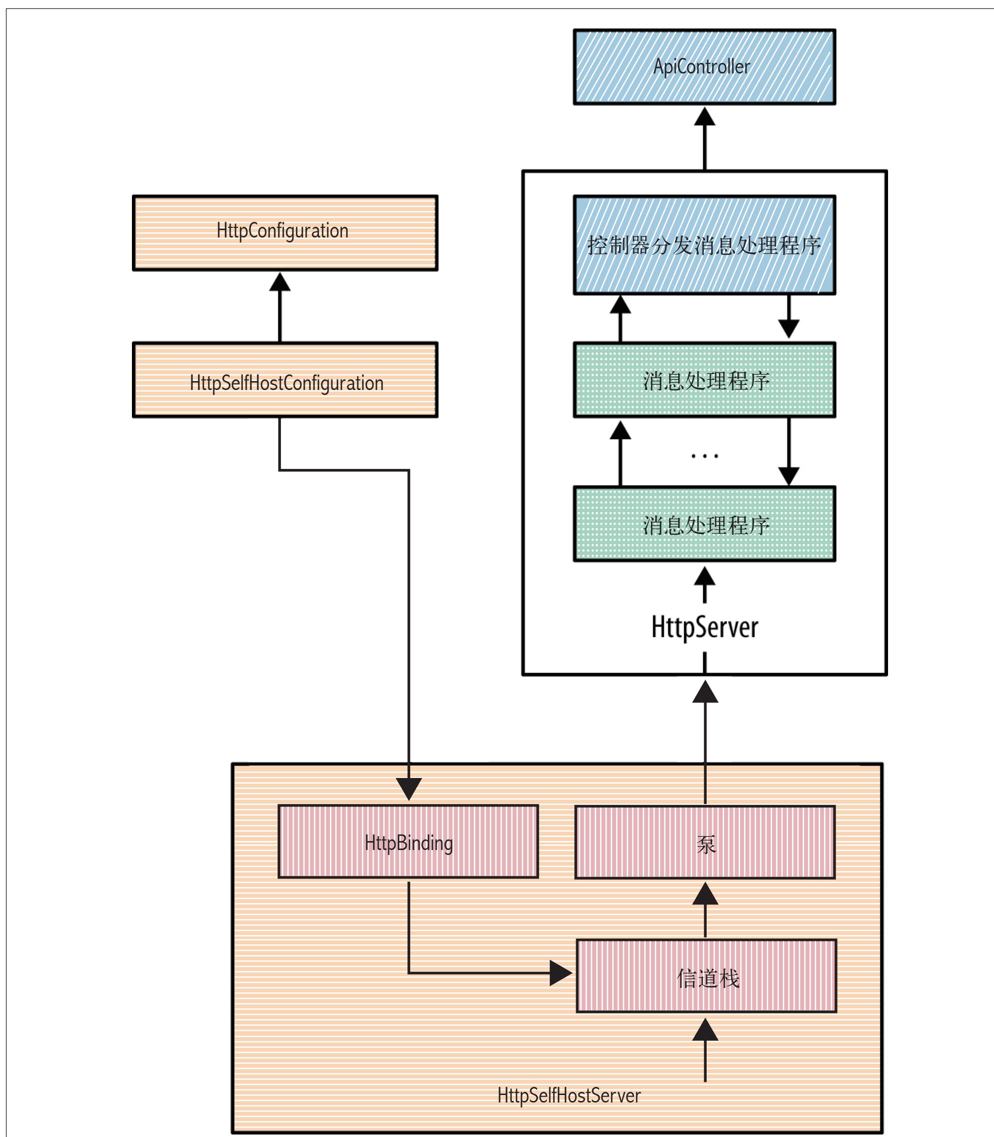


图 11-11：自托管架构

11.2.4 URL预留和访问控制

如果从非管理员账号启动一个自托管 Web 服务器，你通常会得到如下错误：

```
HTTP could not register URL http://+:8080/.
Your process does not have access rights to this namespace
```

为什么会发生这个错误？我们该如何解决这个问题？为了回答这些问题，我们需要对

HTTP 的低层处理架构进行一个简要的介绍。

在 Windows 中，一个内核模式的设备驱动程序，HTTP.sys，监听 HTTP 请求。IIS 和 WCF 自托管传输信道都通过用户模式的 HTTP 服务器 API，使用这个内核模式的驱动。服务器应用程序使用这个 API，注册一个指定 URL 命名空间的请求处理。例如：如果运行示例 11-3，自托管应用程序就会注册 `http://+:8080` 命名空间。主机名中的 + 是一个*强通配符*（strong wildcard），告诉 HTTP.SYS 考虑来自所有网络适配器的请求。但是，这个注册操作受到访问权限的控制，默认情况下，只有具有管理员权限的进程才有权执行这个注册操作。之前看到的错误就是由于这个原因导致的。

要解决这个问题，一个办法是使用管理员权限账号启动自托管应用程序。但是，使用管理员权限运行服务器通常不是一个好主意。更好的解决办法是，赋给运行应用程序的账号所需的权限。为此，我们可以为该账号*预留* URL 命名空间，允许相关应用程序注册预留命名空间里的 URL。我们可以使用命令行工具 `netsh`（这个工具需要管理员权限），进行预留：

```
netsh http add urlacl url=http://+:8080/ user=domain\user
```

其中的 `domain\user` 值应该替换为运行自托管应用程序的用户标识。这个用户标识也可以是 Windows 的特殊账号，例如：`network service` 通常用于运行 Windows 服务中托管的 HTTP 服务器。如果使用 Windows 的特殊账号，`domain\user` 可以替换为 `network service` 或 `local service`。

11.3 用OWIN和Katana托管Web API

在本章开篇，我们就了解到，ASP.NET Web API 是以与宿主无关的方式构建的，这种特性使得 Web API 既可以在传统的 ASP.NET 和 IIS 宿主中运行，也可以在一个定制的进程（自托管）中运行。ASP.NET Web API 与宿主无关的这一特性，给 Web API 的创建和运行创造了新的机会，也带来了新的挑战。例如：在很多情况下，开发者并不希望为了自托管 Web API 而编写一个定制的控制台应用程序或者 Windows 服务。很多具有框架（如 Node.js 或 Sinatra）使用经验的开发者，希望能够将应用程序托管在一个已有的可执行程序中。而且，在如今的 Web 应用程序中，一个 Web API 通常只是多个组件中的一个。其他组件有：服务器端的标记生成框架（如 ASP.NET MVC）、静态服务器文件，以及实时消息框架（如 SignalR）。此外，一个应用程序可能由很多更小的组件构成，每个组件关注具体的任务，例如：认证或者日志。虽然 Web API 现在提供了不同的托管选项，但是一个 Web API 宿主不能同时托管前面提到的其他组件，也就是说，如今的 Web 应用程序中，每个不同的技术都需要有自己单独的宿主。在使用 Web 托管时，IIS 和 ASP.NET 可以请求管道屏蔽这一限制，但是在自托管方式下，这个问题就变得非常明显。

我们实际需要的是一种抽象方式，使很多不同类型的组件形成单个 Web 应用程序，然后使

整个应用程序按照其特定的需求和部署环境，运行在各种服务器和宿主之上。

11.3.1 OWIN

OWIN（Open Web Interface for .NET，.NET 开放 Web 接口，参见 http://owin.org/）是开放源代码社区创建的一个标准，定义了服务器和应用程序组件如何进行交互。创建 OWIN 的目的，是为了改变 .NET Web 应用程序的构建方式——从应用程序作为单个大型框架扩展的方式，转变为小模块松耦合的方式。

为了实现这一目标，OWIN 将服务器和应用程序组件之间的交互简化为一个简单接口，称为应用程序委托或 app func.：

```
Func<IDictionary<string, object>, Task>
```

这个接口是 OWIN 兼容服务器或模块（也称为中间件）需要满足的唯一要求。此外，因为这个应用程序委托只使用了少数几个 .NET 类型，OWIN 应用程序更容易移植到不同版本的框架，甚至不同的平台上，例如：Mono 项目。

这个应用程序委托定义了一个交互，通过这个交互，组件使用一个字典对象（称为环境或环境字典）接收所有状态（包括服务器状态、应用程序状态和请求状态）。因为一个基于 OWIN 的应用程序应该是异步的，所以这个应用程序委托在执行完工作，完成环境字典修改后，返回一个 `Task` 实例。OWIN 标准定义了环境字典中可能或必须存在的几个键 / 值（如表 11-1 所示）。除此之外，任何 OWIN 服务器或宿主可以在这个环境字典中提供自己的条目，这些条目可以供任何其他中间件使用。

表11-1：请求数据的环境条目

必需	键　名	值　说　明
是	`"owin.RequestBody"`	包含请求正文（如果有）的 `Stream`。如果没有请求正文，`Stream.Null` 可以用做占位符
是	`"owin.RequestHeaders"`	请求标头的 `IDictionary<string, string[]>`
是	`"owin.RequestMethod"`	包含请求的 HTTP 请求方法的字符串（例如：`"GET"` 和 `"POST"`）
是	`"owin.RequestPath"`	包含请求路径的字符串。此路径必须是应用程序委托“根”的相对路径
是	`"owin.RequestPathBase"`	字符串，包含请求路径中对应于应用程序委托“根”的部分
是	`"owin.RequestProtocol"`	包含协议名和协议版本的字符串（例如：`"HTTP/1.0"` 或 `"HTTP/1.1"`）
是	`"owin.RequestQuery String"`	字符串，包含 HTTP 请求 URI 的请求字符串部分，不含前导“?”（例如：`"foo=bar&baz=quux"`）。这个值可能为空字符串
是	`"owin.RequestScheme"`	包含请求所用 URI 方案的字符串（例如：`"http"` 和 `"https"`）；参见 URI Scheme

除了将服务器和应用程序的交互定义在这个应用程序委托和环境字典中，OWIN 标准还提供了宿主和服务器实施的一些指导意见，例如：如何处理 URI 和 HTTP 标头，应用程序启动，以及错误处理。这个应用程序委托非常简单，松散类型的环境字典十分灵活，二者结合在一起，使得开发者更容易开发解决专门问题的小型组件，并将其组合为单个应用程序管道。ASP.NET 的下一个版本就将集成几个这样的专门组件，第 15 章会对此进行更多的介绍。

你可以在线获得完整的 OWIN 规范（http://owin.org/spec/owin-1.0.0.html）。

11.3.2 Katana项目

OWIN 规范定义了服务器和应用程序组件如何进行交互，以处理 Web 请求；Katana 则是一组符合 OWIN 规范的组件，这些组件由 Microsoft 开发，作为开源软件发布[1]。如图 11-12 所示，Katana 项目组件按架构层进行组织。图 11-13 展示了流经不同层次和组件的 HTTP 数据。

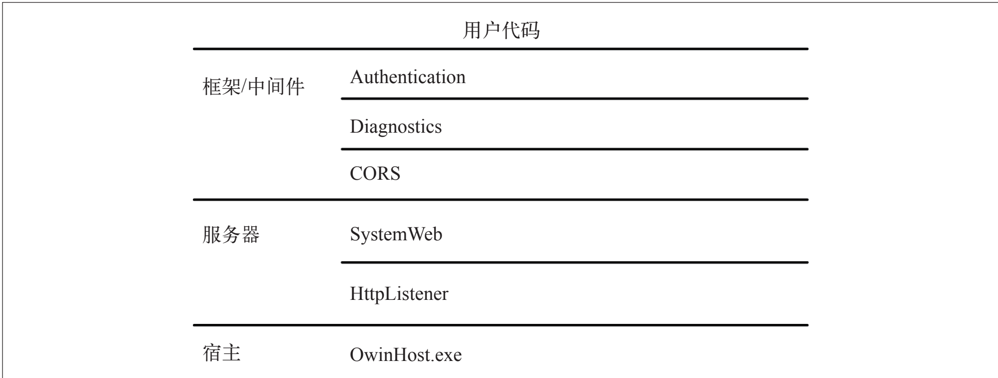


图 11-12：Katana 组件架构以及组件示例

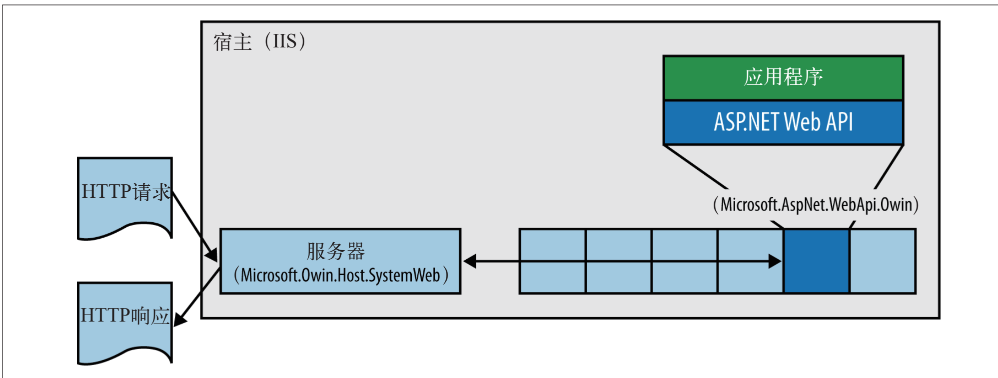


图 11-13：流经运行于 OWIN 管道和 Katana 宿主上的 ASP.NET Web API 应用程序的 HTTP 数据

注 1：除了 Microsoft Katana 组件，许多流行的开源 Web 框架（如 NancyFX、FUBU、ServiceStack 等）都可以运行在 OWIN 管道中。

Kanata 组件分为三层：宿主、服务器和中间件。各层组件的功能如下。

- 宿主

  宿主开始和管理进程。宿主负责启动一个进程，并初始化 OWIN 规范第四节（http://owin.org/spec/owin-1.0.0.html#_4._Application_Startup）中提出的启动序列。

- 服务器

  服务器监听 HTTP 请求，确保环境字典中的值得到正确设置，并为中间件组件管道中的第一个中间件调用其应用程序委托。

- 中间件

  中间件是对请求或响应执行不同任务的组件。这些组件可以执行很小的任务，例如：进行压缩或者实施 HTTPS；也可以作为整个框架（例如：ASP.NET Web API）的适配器。组件组成一个管道结构，其中每个组件都持有指向管道中下一个组件的引用。宿主负责在其启动序列中构建这个管道。

如果以传统的基于框架的方法运行 Web 应用程序，那么宿主、服务器和框架独立于应用程序启动，然后在指定的点调用应用程序。在这种模式中，开发者的代码实际上是底层框架的扩展，因此，应用程序代码对请求处理的控制能力是由框架决定的。而且，在这种模式中，即使应用程序没有使用框架自身中运行的某些功能，也要为其付出性能代价。

在基于 OWIN 的 Web 应用程序中，启动序列与传统方式相反。宿主在初始化环境字典并选择服务器之后，立即发现并调用开发者的应用程序代码，以判断哪些组件应该包含在 OWIN 管道内。在默认情况下，Katana 宿主按照如下规则发现开发者的启动代码：

- （按优先级顺序）查找或发现一个启动类；
- 如果配置文件包含带有 `key` 为 `owin:AppStartup` 的 `appSetting`，则使用此设置值；
- 如果存在程序集属性 `OwinStartupAttribute`，则使用属性中定义的类型；
- 扫描所有程序集，寻找一个名为 `Startup` 的类型；
- 如果找到启动类，找到其中与签名 `void Configuration(IAppBuilder app)` 匹配的方法，调用这个配置方法。

按照这个默认的发现逻辑，我们只需要给项目添加一个启动类，就可以让 Katana 宿主的加载程序发现并运行这个类的配置方法：

```
public class Startup
{
    public void Configuration(IAppBuilder app)
    {
        app.Use(typeof(MyMiddleware));
    }
}
```

在这个启动类的配置方法中，我们可以调用传入的 `IAppBuilder` 对象的 `Use` 方法，构造 OWIN 管道。`Use` 是一个泛型方法，可以将任何实现了应用程序委托的组件配置在管道中。此外，很多中间件组件和框架还提供了自己的扩展方法，以简化管道配置。例如：ASP.NET Web API 提供 `UseWebApi` 扩展方法，可以使用如下代码进行配置：

```
var config = new HttpConfiguration();

// 配置 Web API
// ...

app.UseWebApi(config);
```

但是，当你使用 Web API 的配置扩展方法时，实际发生了什么？通过更深入地了解 Web API 配置方法和 Web API 中间件组件，你可以更好地理解 OWIN 管道和 Katana 的实现，以及 Web API 的宿主适配程序设计的独立性。

11.3.3 Web API配置

`UseWebApi` 方法位于 `System.Web.Http.Owin` 程序集的 `WebApiAppBuilderExtensions` 类中。当从用户的启动类中调用 `UseWebApi` 方法时，这个方法构建 `HttpMessageHanlderAdapter` 类（即 OWIN 的 Web API 中间件组件）的一个实例，并使用泛型方法 `Use`，将其添加到 `IAppBuilder` 实例中。从 `UseWebApi` 方法的代码中，我们可以了解 Katana 基础结构如何将中间件绑定在一起，形成完整的管道：

```
public static IAppBuilder UseWebApi(
    this IAppBuilder builder,
    HttpConfiguration configuration)
{
    IHostBufferPolicySelector bufferPolicySelector =
        configuration.Services.GetHostBufferPolicySelector()
            ?? _defaultBufferPolicySelector;

    return builder.Use(typeof(HttpMessageHandlerAdapter),
        new HttpServer(configuration), bufferPolicySelector);
}
```

泛型方法 `Use` 的第一个参数是 Web API 中间件的类型，其后可以有任意多个附加参数。在添加 Web API 中间件时，代码使用了两个附加参数：一个是 `HttpServer` 实例，用传入的 `HttpConfiguration` 对象进行了配置；另一个对象告诉中间件如何处理请求和响应流。传给 `Use` 方法的中间件不是实例，而是一个类型，因此基础结构在创建中间件实例时，可以（通过中间件的构造函数）对其进行配置，使其带有指向管道中下一个中间件对象的引用。我们可以查看 `HttpMessageHandlerAdapter` 的构造函数，了解具体实现：构造函数的第一个参数是 `next` 引用，其后是传给泛型方法 `Use` 的附加参数。[1]

注 1：第 12 章将详细介绍 Web API 分发逻辑，其中包括 HttpServer 和 HttpMessageHanlder。

```
public HttpMessageHandlerAdapter(OwinMiddleware next,
    HttpMessageHandler messageHandler,
    IHostBufferPolicySelector bufferPolicySelector) : base(next)
```

泛型函数 Use 的输出是修改后的 IAppBuilder 对象，因此扩展方法 UseWebApi 直接返回这个对象。通过这种方式返回 IAppBuilder，我们在启动类中构建 OWIN 管道时，语法可以更为通顺。

11.3.4 Web API中间件

Web API 一旦加入 OWIN 管道，OWIN 服务器就可以调用中间件的应用程序委托，处理 HTTP 请求。让我们回忆一下 OWIN 应用程序委托的签名：

```
Func<IDictionary<string, object>, Task>
```

Web API 的 HttpMessageHandlerAdapter 类的基类 OwinMiddleware（位于 Microsoft.Owin NuGet 软件包中）直接提供这一函数。基类 OwinMiddleware 为服务器提供了应用程序委托函数，并为其派生类提供了一个更简单的 API：

```
public async override Task Invoke(IOwinContext context)
```

其中的上下文对象提供一个类型更强的对象模式，用于访问环境字典成员，例如：HTTP 请求和响应对象。表 11-2 概括了 IOwinContext 目前提供的访问方法列表。

表11-2：IOwinContext的属性访问方法

请求	对当前请求的封装方法
响应	对当前响应的封装方法
环境	封装的 OWIN 环境字典
认证（.NET 4.5 及更高版本）	访问当前请求可以使用的认证中间件功能

上下文对象中的每个属性提供对环境字典中不同成员的强类型访问。要了解每个不同的封装类型，请参考 Microsoft.Owin 的源代码（http://katanaproject.codeplex.com/SourceControl/latest#README）。

在流经 OWIN 管道时，如果请求遇到了 HttpMessageHandlerAdapter 的 Invoke 方法，会按照图 11-14 所示的数据流程进行处理。

作为 OWIN 管道和 Web API 编程模型之间的桥梁，HttpMessageHandlerAdapter 执行的一个动作是将在 OWIN 环境字段中找到的对象转换为 Web API 使用的基础类型。当然，这些基础类型就是 HttpRequestMessage 和 HttpResponseMessage。在将 HTTP 请求发送给 Web API 进行处理之前，中间件还会从环境字典中，（通过 IOwinContext.Request.User）取得用户对象（如果存在的话），并将这个对象赋给活动线程的 CurrentPrincipal 属性。

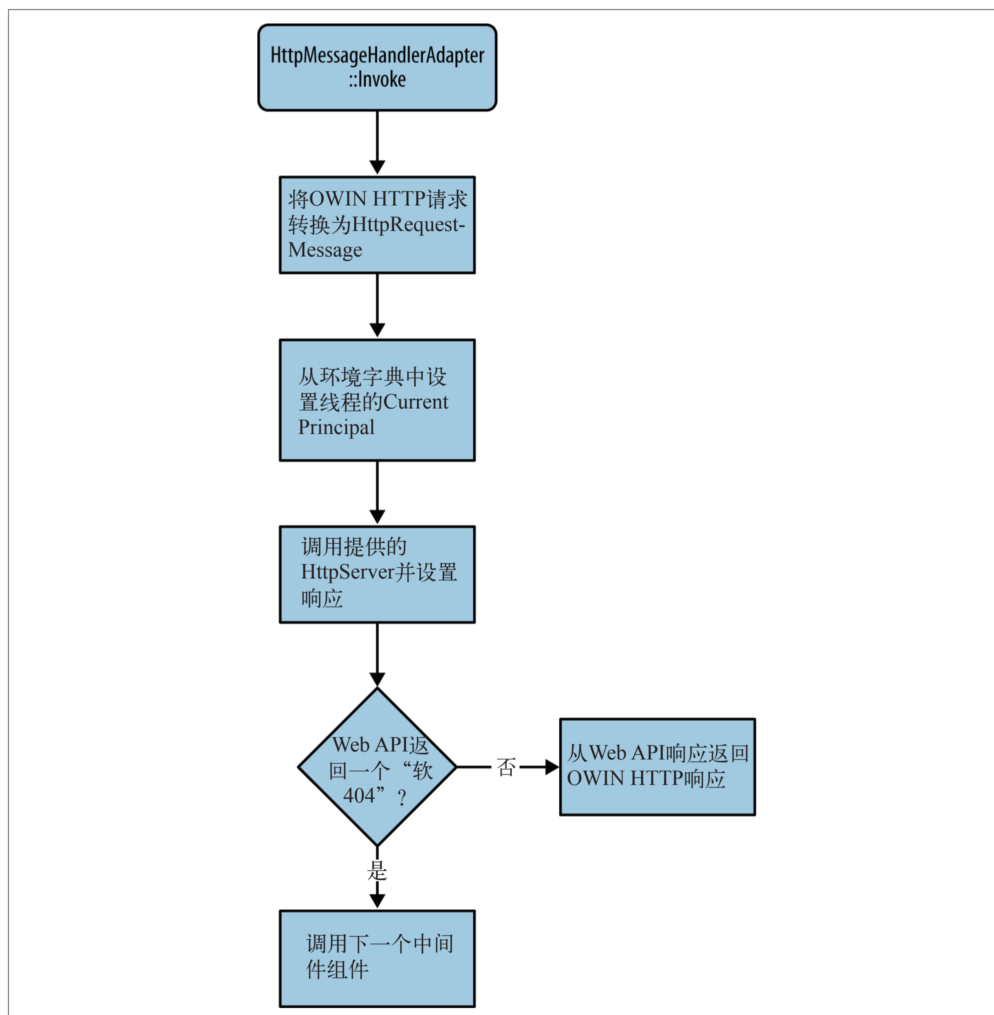


图 11-14：Web API 中间件数据流

如果中间件得到了请求的 `HttpRequestMessage` 表示，就可以采取与之前描述的 Web API 托管基础结构组件类似的方式，调用 Web API。第 12 章将介绍，`HttpServer` 类型派生自 `HttpMessageHandler`，是 Web API 消息处理程序管道的入口（开发者也可以通过扩展方法重载，指定另一个称作分发程序的 `HttpMessageHandler` 对象，作为消息处理程序管道的最后一个节点）。`HttpMessageHandler` 不能直接进行调用，因此中间件将其封装在一个 `HttpMessageInvoker` 对象中，使用如下代码调用：

```
response = await _messageInvoker.SendAsync(request, owinRequest.CallCancelled);
```

这个调用开始了 `HttpRequestMessage` 在 Web API 的消息处理程序管道和控制器管道中的处

理过程，并将指向结果 `HttpResponseMessage` 的引用保存在一个本地变量中。第 12 章将详细讨论消息处理程序和控制器管道。

Web API 中间件的一个额外任务是，判断如何处理 `HttpResponseMessage` 的 HTTP 状态码 `404 Not Found`。这个任务非常重要，因为在 OWIN 管道的环境中，404 状态码的产生可能有以下两种原因。

- 请求不匹配 `HttpConfiguration` 对象中指定的任何 `HttpRoutes`。在这种情况下，中间件应该调用下一个中间件组件的应用程序委托。
- 作为应用程序协议的实现，应用程序开发者明确返回这个状态码（例如，对于请求 `GET /api/widgets/123`，小工具数据库中无法找到物品 123）。在这种情况下，中间件不应该调用组件链中的下一个组件，而是应该向客户端返回状态码为 `404` 的响应。

对于 Web API 中间件而言，由 Web API 的路由匹配逻辑设置的 `404` 响应码称为一个“软 404"，可以通过响应消息属性集中的一个附加设置——`HttpPropertyKeys.NoRouteMatched`——存在与否来判断。如果不存在这个设置，中间件认为一个 `404` 响应码是“硬 404”，会立刻向客户端返回一个状态码为 `404` 的 HTTP 响应。

11.3.5　OWIN生态环境

Katana 组件的内容比本章中讨论的要多很多。最新发布的 Katana 组件包含了认证组件（既有社会开发，也有企业提供的中间件）、诊断中间件、`HttpLisenter` 服务器，以及 `OwinHost.exe` 宿主。第 15 章将更为详细地介绍基于 OWIN 的认证组件。随着时间推移，Microsoft 将会提供更多兼容 OWIN 的组件，逐步覆盖 `System.Web.dll` 目前提供的许多常用功能。此外，由社区创建的第三方组件也在持续增加，目前已经包含了很多不同的 HTTP 框架以及中间件组件。在未来几年，我们将会看到 OWIN 组件空间的急剧扩展。

11.4　内存托管

还有一种 Web API 托管方式主要在测试时使用，这种托管方式基于 `HttpClient` 实例和 `HttpServer` 实例的直接连接，通常称为*内存托管*（in-memory hosting）。

第 14 章将会介绍，`HttpClient` 实例可以由构造函数中传入的 `HttpMessageHandler` 进行配置。客户端随后可以使用这个 `HttpMessageHandler`，从 HTTP 请求异步获得 HTTP 响应。通常情况下，这个 `HttpMessageHandler` 要么是一个 `HttpClientHandler`，使用底层网络基础结构，发送和接收 HTTP 消息；要么是一个 `DelegatingHanlder`，对请求和响应各自执行前后处理。

但是，`HttpServer` 类也可以对 `HttpMessageHandler` 进行扩展，也就是说，在构造 `HttpClient` 时，你也可以使用 `HttpServer`。这使得客户端和服务器可以直接进行内存通信，无需任何网络栈的开销，这个功能在测试中非常有用。示例 11-5 展示了如何使用内存托管功能。

示例 11-5：内存托管

```
var config = new HttpConfiguration();
config.Routes.MapHttpRoute("default", "{controller}/{id}",
                                new { id = RouteParameter.Optional });
var server = new HttpServer(config);
var client = new HttpClient(server);
var c = client.GetAsync("http://can.be.anything/resource").Result
    .Content.ReadAsStringAsync().Result;
```

示例 11-5 中的主机名 `can.be.anything` 一点也没有夸张，因为没有使用网络层，系统会忽略 URI 的主机名部分，所以主机名可以随便写。

`HttpServer` 和 `HttpClient` 是对称的，一个是消息处理程序，另一个接收消息处理程序，因此客户端和服务器可以建立直接联系，如图 11-15 所示。

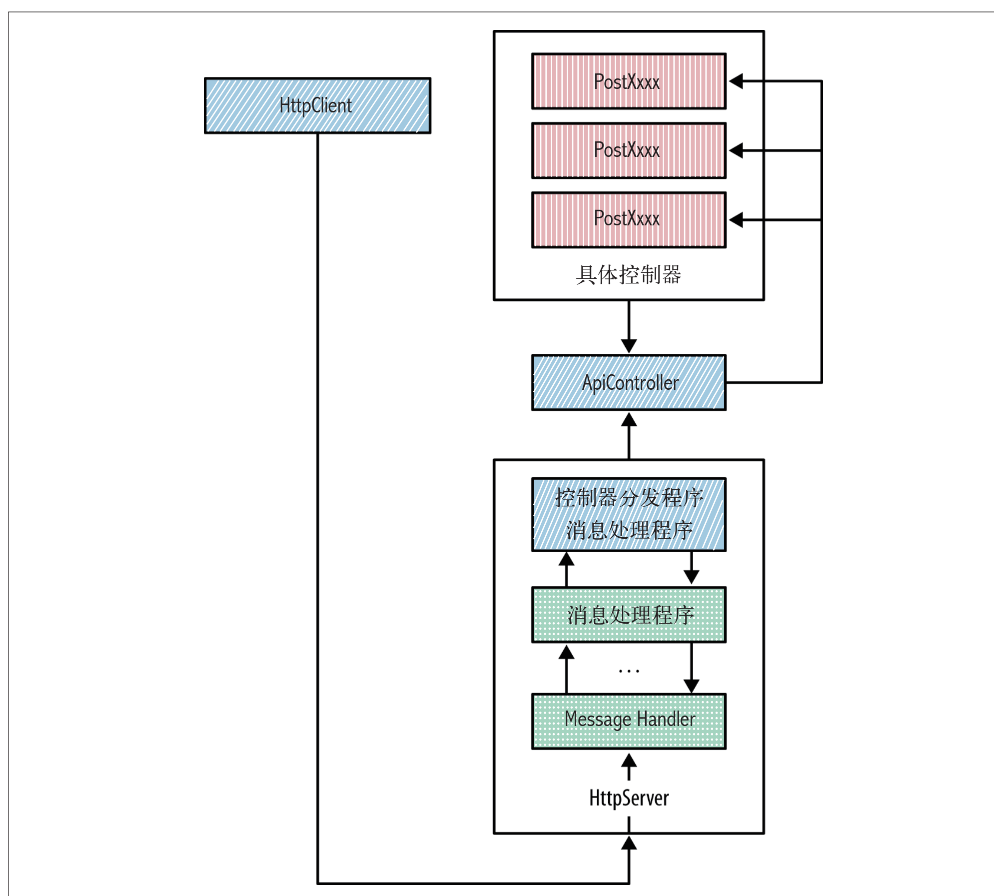


图 11-15：内存托管示意图

11.5 Azure Service Bus Host

在本章最后，我们要举例介绍开发定制托管适配层。我们将以 Windows Azure Service Bus 为例，这是一个云托管的基础结构，提供消息中转（brokered）和中继（relayed）功能。消息中转机制有：队列（queue）和主题（topic），既使发送者和接受者在时间上去耦，又提供消息广播。消息中继可以将托管在内部网络中的 API 向外部公开，是这一节讨论的重点。

在图 11-16 中，一个 API（即：一组资源）托管在一台具有如下特征的机器上：

- 位于内部网络中，没有外部 IP 或外部 DNS 名；
- 通过 NAT（Network Address Translation，网络地址转换）和防火墙系统，与因特网隔离。

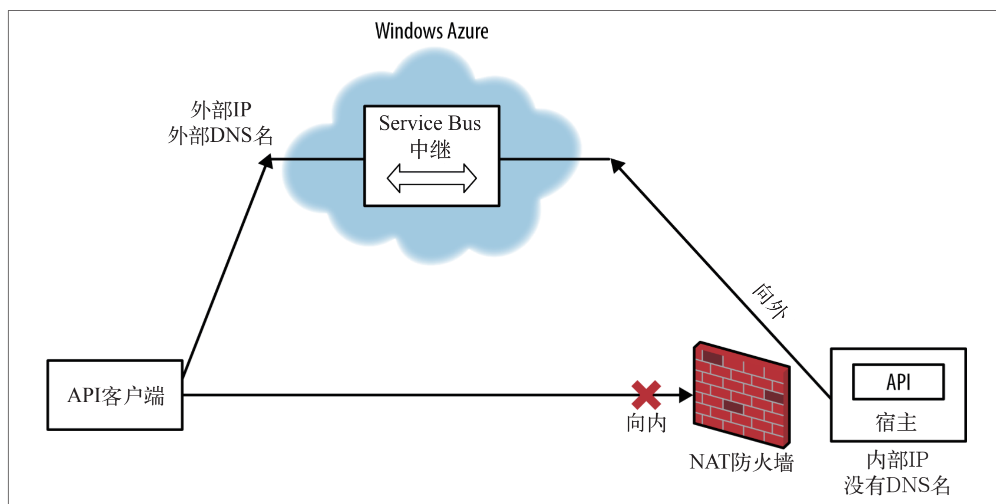


图 11-16：Service Bus 使用场景

这种设定的一个具体例子是，提供 Web API 的家居自动化系统。通常的家庭因特网连接（例如，通过 DSL 连接）具有图 11-16 中描述的特征——即：没有外部 IP 地址或 DNS 名，也没有 NAT 和防火墙阻断进入的连接。但是，如果位于因特网上的外部客户端可以使用这个 API，这个功能将非常有用。例如：设想一下，我们可以据此使用智能手机，远程控制房间温度，或者查看监控画面。

如图 11-16 所示，Service Bus 中继功能作为客户但和 API 宿主之间的中介，解决了这些连接问题。

- 首先，宿主与 Service Bus 中继建立一个向外的连接。因为这个连接是向外的，不是向内的，所以在内部不需要公共 IP，网络地址的转换由 NAT 完成。
- 建立连接之后，Service Bus 中继使用自己的命名空间中的一个域名（例如：`webapibook.servicebus.windows.net`），创建和提供一个外部端点。

- 发送到这个外部端点的每个请求，都会通过之前建立的向外连接，接着发送到 API 宿主。由 API 宿主产生的响应也通过这个向外连接返回，并由 Service Bus 中继发送给客户端。

Azure Service Bus 支持多个租户，每个租户拥有一个 DNS 名，格式为 `{tenant-namespace}.servicebus.windows.net`。例如：这一节的示例中 DNS 名为 `webapibook.servicebus.windows.net`。当宿主与 Service Bus 建立连接，告诉中继服务开始监听请求时，宿主必须进行认证——也就是说，要证明自己可以使用这个租户的名字。而且，宿主必须定义一个路径前缀，用于与租户的 DNS 结合，形成基地址。只有带有这个前缀的请求，中继服务才会将其转发给宿主。

Azure Service Bus 提供一个 SDK（Software Development Kit，软件开发工具包），可以与 WCF 编程模型集成，为使用 Service Bus 中继的宿主服务提供特殊的绑定。遗憾的是，在这本书编写时，这个工具包还不支持 ASP.NET Web API。但是，基于 Web API 的托管独立功能，参考基于 WCF 的自托管方式，我们可以构建定制的 `HttpServiceBusServer` 类，使用 Service Bus 中继服务来进行 ASP.NET Web API 托管。

图 11-17 展示了 `HttpServiceBusServer` 托管服务器以及相关的类。

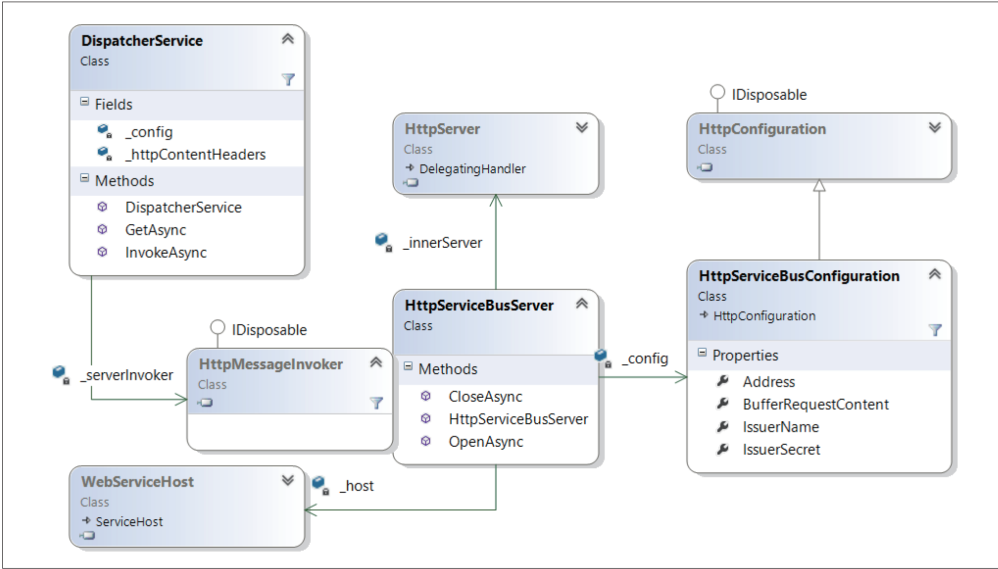


图 11-17：`HttpServiceBusServer` 及其相关类

这个新的服务器使用 `HttpServiceBusConfiguration` 类（派生自基类 `HttpConfiguration`）的一个实例进行配置，并专门为这种托管情况添加了如下属性：

- 外部的 Service Bus 中继地址（例如：`https://tenantnamespace.servicebus.windows.net/some/path`）；

- 与 Service Bus 中继建立向外连接所需的身份凭证。

HttpServiceBusServer 使用了派生自基类 HttpConfiguration 的专门配置类，这种设计与自托管方式（参见图 11-8）类似。HttpServiceBusServer 在内部创建一个 WCF WebServiceHost，添加一个由 WebHttpRelayBinding 配置的端点。WebHttpRelayBinding 是 Service Bus SDK 提供的新的绑定类，与 WCF 本地的 WebHttpBinding 类似，主要的不同之处在于：服务是通过 Service Bus 中继远程提供的，而不是在本地的托管机器上。通过端点接收到所有请求都由 DispatcherService 类的一个实例进行处理。

```
[ServiceContract]
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
                        ConcurrencyMode = ConcurrencyMode.Multiple)]
internal class DispatcherService
    [WebGet(UriTemplate = "*")]
    [OperationContract(AsyncPattern = true)]
    public async Task<Message> GetAsync()
    {...}

    [WebInvoke(UriTemplate = "*", Method = "*")]
    [OperationContract(AsyncPattern = true)]
    public async Task<Message> InvokeAsync(Message msg)
    {...}
}
```

这个泛型服务实现了两个异步操作：Get 和 Invoke。Get 操作处理所有 GET 方法的 HTTP 请求。Invoke 操作处理其他的所有请求（Method = "*"）。请注重，这两个操作都有 UriTemplate = "*"，说明都处理任何路径的请求。

当这两个方法中的任何一个收到一个请求时，请求消息的本地表示会转换成一个新的 HttpRequestMessage 实例，然后这个新实例会推送到一个内部的 HttpServer。这个内部 HttpServer 在 HttpServiceBusServer 构造函数中创建，由传入的 HttpServiceBusConfiguration 配置。可是，HttpServer.SendAsync 是一个受保护方法，因此不能直接调用。但是，HttpMessageInvoker 可以封装包括 HttpServer 在内的任何消息处理程序，并提供一个公共的 SendAsync 方法：

```
public DispatcherService(HttpServer server, HttpServiceBusConfiguration config)
{
    _serverInvoker = new HttpMessageInvoker(server, false);
    _config = config;
}
```

当 HttpServer 生成 HttpResponseMessage 后，DispatcherService 将 HttpResponseMessage 转换回 WCF 消息表示并返回。

HttpServiceBusServer 的整体设计受到基于 WCF 的自托管适配层的启发，但有两点不同之处。第一个，也是最重要的不同之处在于，Service Bus 宿主位于 WCF 服务模型之上，而

自托管直接使用了 WCF 信道。这种设计虽然会带来额外的开销，但是实现更为简单，因而得到采纳。

第二个不同之处是，HttpServiceBusServer 没有直接继承 HttpServer 类。HttpServiceBusServer 没有像 HttpSelfHostServer 一样使用基于类继承的设计，而是使用组合方式：在内部创建和使用一个 HttpServer 实例。

HttpServiceBusServer 的实现可以在源代码（https://github.com/pmhsfelix/WebApi.Explorations.ServiceBusRelayHost）中找到。源代码库还提供一个示例，展示使用这个新的 Web API 托管方式的简单性。ServiceBusRelayHost.Demo.Screen 项目定义了一个 Service Bus 托管的服务，其中只有一个资源。

```
public class ScreenController : ApiController
{
    public HttpResponseMessage Get()
    {
        var content = new StreamContent(ScreenCapturer.GetEncodedByteStream());
        content.Headers.ContentType = new MediaTypeHeaderValue("image/jpeg");
        return new HttpResponseMessage()
        {
            Content = content
        };
    }
}
```

代码中用到的 ScreenCapturer 是一个辅助类，用于屏幕截图。这个资源控制器的托管也非常简单。

```
var config = new HttpServiceBusConfiguration(
    ServiceBusCredentials.ServiceBusAddress)
{
    IssuerName = "owner",
    IssuerSecret = ServiceBusCredentials.Secret
};
config.Routes.MapHttpRoute(
    "default",
    "{controller}/{id}",
    new { id = RouteParameter.Optional });
var server = new HttpServiceBusServer(config);
server.OpenAsync().Wait();
...
```

首先，代码使用 Service Bus 地址、访问凭据（IssueSecret）和访问用户名（owner）初始化一个 HttpServiceBusConfiguration 实例；然后，和其他的托管方式一样，将路由添加到 Routes 属性；最后，使用这个配置实例，配置一个 HttpServiceBusServer，并明确启动服务器。

图 11-18 展示了通过一个简单的旧浏览器，访问这个通过 Azure Service Bus 托管的屏幕资

源的结果。请注意，浏览器的地址栏中使用了一个外部 DNS 名。

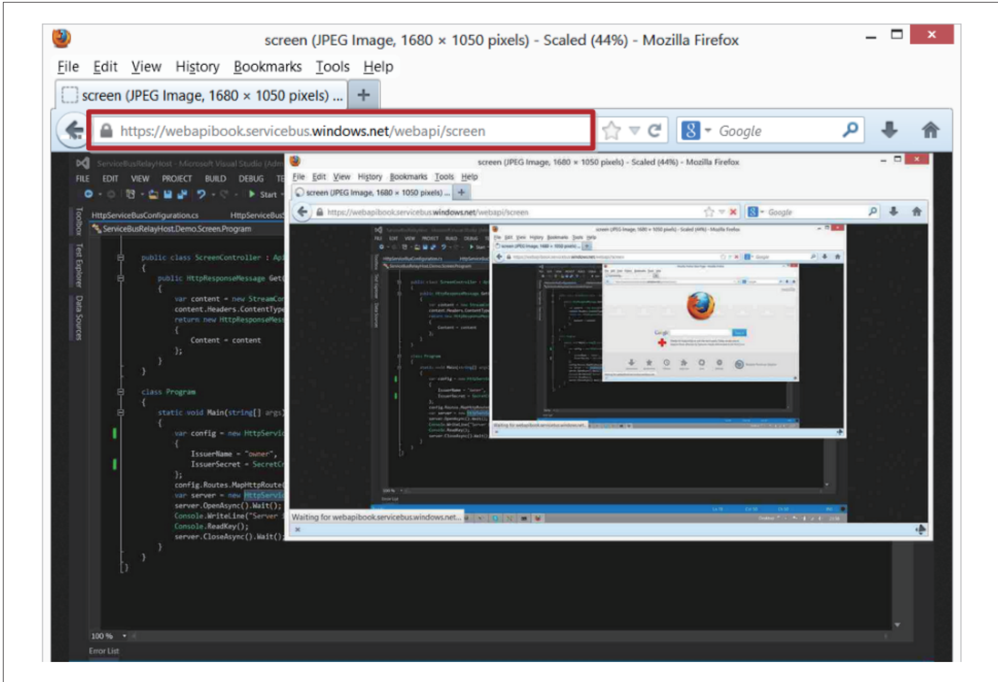


图 11-18：访问通过 Service Bus 托管的屏幕资源

11.6　小结

本章主要关注 Web API 与外部的托管基础结构进行交互的方式，不仅描述了已有的托管适配层（Web 托管和自托管），而且讨论了基于 OWIN 规范的新托管方式和 Katana 项目，最后还介绍了内存托管和定制托管适配层的一个例子。在随后的章节中，我们将转而关注 ASP.NET Web API 中更高的层次，特别是路由和控制器。

第12章

控制器和路由

> 知道家里的水管如何工作，大部分时候都没什么用处——但是当你需要这些知识时，一无所知的后果就会很严重。

虽然 ASP.NET Web API 提供了许多有用的高层功能，从序列化和模型绑定到 OData 风格查询支持，但是和所有的 Web API 一样，ASP.NET Web API 的核心任务是处理 HTTP 请求并提供适当的响应。因此，对于一个 HTTP 请求如何从客户端开始，流经 ASP.NET Web API 基础结构和编程模型的各个元素，最终产生一个可以发送回客户端的 HTTP 响应，我们需要理解这一过程的核心机制，这一点至关重要。

本章关注前面提到的消息流，讨论其基本机制，以及支持请求处理和响应生成的编程模型。此外，本章还将介绍关键类型和插入点（insertion point）。使用这些插入点，我们可以扩展 ASP.NET Web API 框架，支持定制的消息流和处理方案。

12.1 HTTP消息流概览

消息流经 ASP.NET Web API 的确切过程，会随着托管方式的不同而略有变化，托管在第 10 章中进行了详细的介绍。但是，参与 HTTP 消息流的框架组件大致可分为两类（参见图 12-1）：

- 依靠 HTTP 消息获得上下文的组件；
- 依靠高层编程模型获得上下文的组件。

第一类组件只依靠来自底层“消息处理程序”管道的核心 HTTP 消息上下文。这些组件从托管抽象层获得一个 `HttpRequestMessage` 对象，并最终负责返回一个 `HttpResponseMessage` 对象。

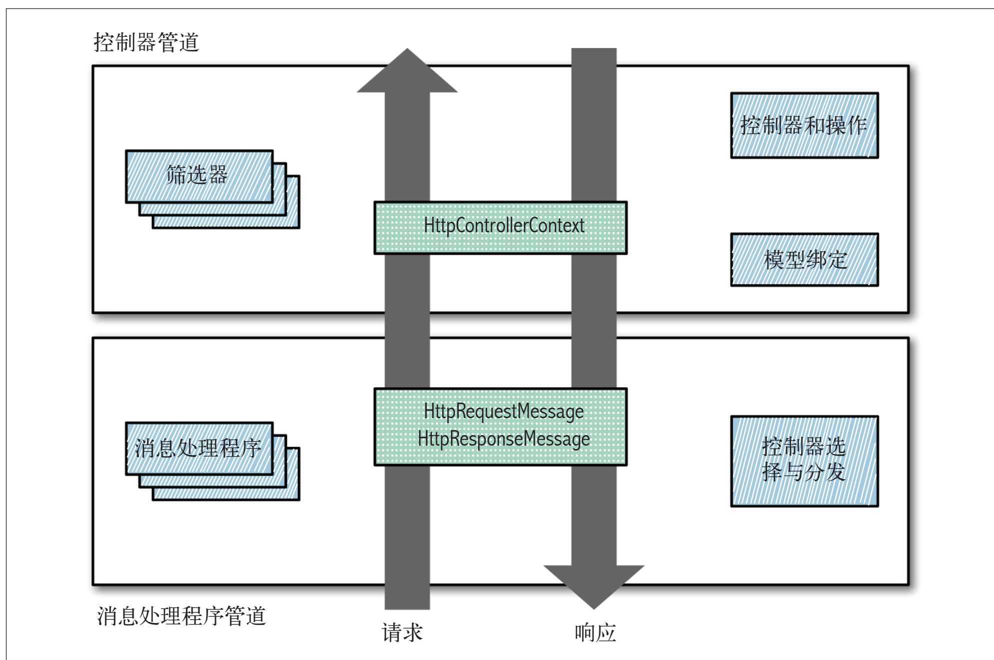


图 12-1：消息处理程序和控制器管道

依赖高层编程模型的组件则不同，这些组件可以访问并使用编程框架抽象层，例如：控制器和操作方法，以及映射到 HTTP 请求不同元素的参数。

如前所述，像选择 URL 路由这样的低层机制，会随着托管方式的不同而变化。例如，如果 Web API 作为托管在 IIS 上的 MVC 应用程序的一部分进行托管，那么 HTTP 消息会流经 ASP.NET 提供的核心路由基础结构。而自托管 Web API 时，消息会流经以 `HttpListener` 对象为核心构建的 WCF 信道栈。无论选择何种托管方式，一个请求最终都会转换成一个 `HttpRequestMessage` 实例，传给一个 `HttpServer` 实例。

12.2 消息处理程序管道

对于宿主相关的组件，`HttpServer` 是消息处理程序管道的入口。`HttpServer` 调用 `HttpClientFactory` 的 `CreatePipeline` 方法，使用全局和路由配置数据中提供的处理程序，初始化管道。其代码如示例 12-1 所示。

示例 12-1：初始化消息处理程序管道

```
protected virtual void Initialize()
{
    // 进行配置的最后初始化
    // 从这里开始，系统认为配置不可变
    _configuration.Initializer(_configuration);
```

```
        // 创建管道
        InnerHandler = HttpClientFactory.CreatePipeline(_dispatcher,
            _configuration.MessageHandlers);
    }
```

最后，因为 HttpServer 自己派生于 DelegatingHanlder 类，HttpServer 成为了消息处理程序管道中的第一个处理程序。整个管道首先是 HttpServer 以及其后的任意多个定制的 DelegatingHanlder 对象组成，这些定制对象注册在 HttpConfiguration 中；接下来是另外一个特殊的处理程序 HttpRoutingDispatcher；管道的最后，要么是一个在路由注册时提供的路由相关的定制消息处理程序（或者是由 HttpClientFactory.CreatePipepline 构建的另一个消息处理器管道），要么是默认的 HttpControllerDispatcher 消息处理程序。HttpControllerDispatcher 选择和创建一个控制器实例，并将消息分发到这个实例。图 12-2 展示了整个管道的组成。

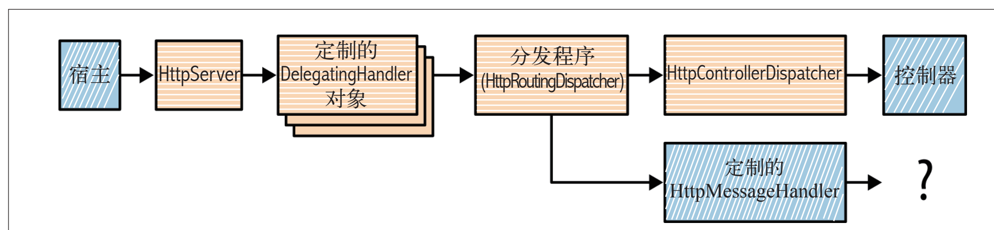


图 12-2：消息处理程序管道

HttpServer 将 HttpClientFactory.CreatePipeline 返回的值赋给自己的 InnerHandler 属性，从而成为管道的第一个节点。因此，HttpServer 可以调用基类的 SendAsync 方法，将控制权移交给管道中的下一个处理程序。管道中所有的消息处理程序都使用这种方式移交控制权。

```
    return base.SendAsync(request, cancellationToken)
```

基类 DelegatingHanlder 直接调用对象的 InnerHandler 的 SendAsync 方法。对象的内部处理程序在自己的 SendAsync 方法中处理消息，然后重复这一过程，调用自己的内部处理程序的 SendAsync 方法。这个过程一直持续到最后一个处理程序——对于典型的 ASP.NET Web API，最后一个就是将请求分发到控制器实例的处理程序。这种风格的管道（如图 12-3 所示）有时称作“俄罗斯套娃”，因为处理程序一个套着一个，外部处理程序直接调用其内部的处理程序，使得请求数据从最外层的处理程序一直流向最内层的处理程序（然后响应数据反向流出）。[1]

注 1：有一种风格是，管道在组件之外，管道调用一个组件，获得响应，然后调用下一个组件，以此类推，使得数据在管道中流动。这种风格可以与“俄罗斯套娃”风格进行对比。

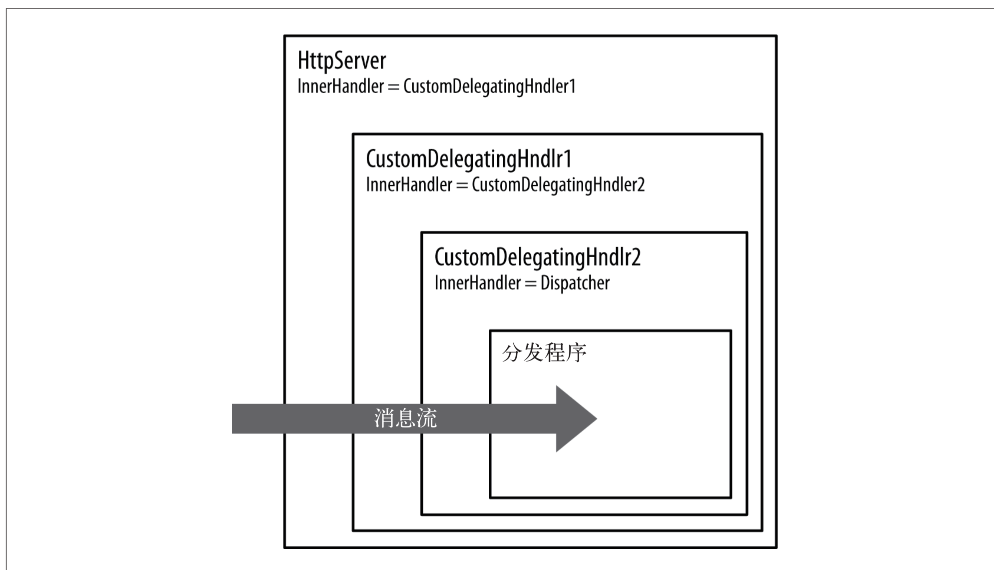


图 12-3：消息处理程序的“俄罗斯套娃”模型

请记住：这整个数据流都是异步的，因此从 SendAsync 返回的值是 Task。实际上，SendAsync 的完整签名是这样的：

```
Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
    CancellationToken cancellationToken)
```

你可能需要一段时间，才能逐步适应基于任务的异步管道的使用，第 10 章详细介绍了相关的内容。但是，创建基于任务的消息处理程序有如下一些基本规则。

- 要将控制权传递给管道中的下一个，或内部的处理程序，你可以返回调用基类 SendAsync 方法的返回值。
- 要终止进一步的消息处理，返回一个响应（也称为“短路”请求处理），你可以返回一个新的 Task<HttpResponseMessage>。
- 要在 HTTP 响应从最内部处理程序流回最外部处理程序时进行处理，你可以在返回的任务上附加一个延续（continuation，用 ContinueWith 方法实现）。这个延续应该只有一个参数，即需要延续的任务，并应该返回一个 HttpResponseMessage 对象。在 .NET 框架 4.5 及更高版本中，你可以使用 async 和 await 关键字，简化异步代码的实现。

请参考示例 12-2 中的消息处理程序，这个处理程序查看一个 HTTP GET 请求，判断这个请求是否为一个条件 GET（即：包含一个 if-non-match 标头的请求）。如果这个请求是条件请求，并且本地缓存中没有这个请求的实体标签（ETag），处理程序就认为底层的资源状态值发生了改变。因此，处理程序让这个请求继续流经管道，通过调用 base.SendAsync 并返回其结果，将请求传递给适当的处理程序。这样可以保证这个 GET 请求的响应包含资源的最新表示。

示例 12-2：处理带有 ETag 的条件 GET 请求的消息处理程序

```
protected override Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request,
    CancellationToken cancellationToken)
{
    if (request.Method == HttpMethod.Get &&
        request.Headers.IfNoneMatch.Count > 0 &&
        (!IfNoneMatchContainsStoredEtagValue(request)))

        return base.SendAsync(request, cancellationToken).ContinueWith(task => {
            var resp = task.Result;
            resp.Headers.ETag = new EntityTagHeaderValue(
                _eTagStore.Fetch(request.RequestUri));
            return resp;
        });
    }

    ...

    // 默认情况下，让请求继续流经消息处理程序管道
    return base.SendAsync(request, cancellationToken);
}
```

这个处理程序也可以给返回的任务添加一个延续，以便给响应消息创建和设置一个新的 ETag 值。此后对这个资源的请求就可以传递这个新的 ETag 值，进行验证。

使用 HttpMessageInvoker 调用一个消息处理程序

如果你使用过消息处理程序，可能已经知道 SendAsync 方法是一个受保护的内部方法，因此可能会问，HttpServer 本身是从 DelegationHandler 派生的，外部类型（也就是组成底层托管基础结构的类型）怎么能对 HttpServer 的 SendAsync 进行调用呢？

为了调用 SendAsync 方法，类可以使用 System.Net.Http 程序集提供的 HttpMessageInvoker。Sytem.Net.Http 程序集还提供 DelegatingHandler 的基类 HttpMessageHandler。

因为 HttpMessageInvoker 和 HttpMessageHandler 位于同一个程序集中，而且 SendAsync 是一个受保护的内部方法，因此 SendAsync 可以从 HttpMessageHandler 的派生类或者同一个程序集中的类型中调用，位于同一程序集中的 HttpMessageInvoker 就可以调用 SendAsync。因此，要执行一个消息处理程序，你可以构建一个新的 HttpMessageInvoker，用如下代码调用其公开的 SendAsync 方法：

```
var handler = new MyHandler();
var invoker = new HttpMessageInvoker(handler);

Task responseTask = invoker.SendAsync(request, cancellationToken);
```

12.2.1 分发程序

消息处理程序管道的最后一个阶段是分发。在 ASP.NET Web API 的较早版本中，分发阶段是预先定义的，从路由数据提供的信息中选择一个控制器，获得该控制器的实例，然后把 HTTP 消息和上下文信息传给控制器，由控制器的执行逻辑进行处理。你还是可以添加一个定制的消息处理器，返回一个新的 Task 对象，从而绕过控制器编程模型。但是，这个消息处理器必须添加到全局的 HttpConfiguration 对象中，也就是说，这个处理器需要处理发送到这个 Web API 的每一个 HTTP 请求。

为了使消息处理程序可以按不同路由进行配置，也为了支持使用非 IHttpController 的高层抽象的不同的 Web API 框架，ASP.NET Web API 团队给分发过程添加了一个间接层。HttpServer 使用 HttpRoutingDispatcher 的一个实例作为消息处理程序管道的最后一个节点。从下面的产品源代码节选，我们可以看到，HttpRoutingDispatcher 负责调用由路由提供的一个定制的消息处理程序，或者默认的 HttpControllerDispatcher。HttpControllerDispatcher 派生自 HttpMessageHandler，HttpMessageHandler 无法直接调用，因此 HttpRoutingDispatcher 将分发程序实例封装在一个 HttpMessageInvoker 对象中执行：

```
var invoker = (routeData.Route == null || routeData.Route.Handler == null) ?
    _defaultInvoker : new HttpMessageInvoker(routeData.Route.Handler,
    disposeHandler: false);
return invoker.SendAsync(request, cancellationToken);
```

路由相关的消息处理程序作为路由配置自身的一部分进行声明。例如，请看下面的路由注册代码：

```
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute("customHandler", "custom/{controller}/{id}",
        defaults: new {id = RouteParameter.Optional},
        constraints: null,
        handler: HttpClientFactory.CreatePipeline(
                    new HttpControllerDispatcher(config),
                    new[] {new MyHandler()})
    );

    ...
}
```

customHandler 路由除了包含标准的路由配置和注册代码，还提供一个定制的消息处理程序，作为 MapHttpRoute 的最后一个参数。但是，实际上，这段代码不仅注册了定制消息处理程序 MyHandler 的一个实例，还使用帮助方法 HttpClientFactory.CreatePipeline，用默认的 HttpControllerDispatcher 消息处理器构建 MyHandler。如果你要插入路由相关的消息处理程序，一定要记住这一点：如果将一个定制消息处理程序提供给 HttpRoutingDispatcher，这个消息处理程序就要负责处理将来所有的 HTTP 消息。CreatePipeline 的第一个参数是所需的“最终目标”消息处理程序，后面的参数是组成管道的其他所有消息处理程

序。CreatePipeline 方法会为每个消息处理程序设置 InnerHandler 属性，将其连成一串，并返回串中的第一个消息处理程序。在示例中，这个“串”包含 MyHandler 以及后面的 HttpControllerDispatcher。请记住：对于这样创建的消息处理程序管道，除了最内部的处理程序之外，其他所有消息处理程序都必须派生自 DelegatingHanlder，不能直接继承 HttpMessageHandler，因为 DelegatingHandler 才支持通过 InnerHandler 属性进行连接。

12.2.2 HttpControllerDispatcher

在默认情况下，消息处理程序管道的最后一环是 HttpControllerDispatcher。这个处理程序将消息处理程序管道和上层的编程模型元素控制器和操作（我们称之为控制器管道）绑定在一起。HttpControllerDispatcher 执行三项任务：

- 使用一个实现 IHttpControllerSelector 接口的对象，选择一个控制器；
- 使用一个实现 IHttpControllerActivator 接口的对象，获得一个控制器的实例；
- 给控制器实例传入一个控制器上下文对象，其中包含当前配置、路由和请求上下文，执行控制器实例。

为了完成这些任务，HttpControllerDispatcher 使用了两种值得注意的类型，即实现 IHttpControllerSelector 接口的类型，和实现 IHttpControllerActivator 接口的类型。

12.2.3 控制器选择

正如其名称所示，IHttpControllerSelector 的作用是基于 HTTP 请求选择适当的控制器。ASP.NET Web API 提供一个默认实现 DefaultHttpControllerSelector。这个类使用如下算法进行控制器选择。

- 判断控制器是否可以从路由数据直接发现。使用基于属性的路由时，这一条件为真。
- 检查控制器名是否有效。如果控制器名缺失或者为空字符串，就抛出一个 404 响应异常。
- 使用控制器名，在控制器信息缓存中寻找匹配的 HttpControllerDescriptor 并返回。

控制器信息缓存是一个字典，保存控制器名，以及缓存首次访问时初始化的 HttpControllerDescriptor 对象。在初始化过程中，控制器信息缓存使用 HttpControllerTypeCache 的一个实例，该实例使用实现 IHttpControllerTypeResolver 接口的一个对象，遍历程序集和类型，构建包含所有有效控制器类型的列表。默认情况下，Web API 使用 DefaultHttpControllerTypeResolver。DefaultHttpControllerTypeResolver 认为符合如下条件的类型为有效的控制器：

- 是一个类；
- 是公共类；
- 是非抽象类；

- 实现或继承一个实现 IHttpController 接口的类；
- 类名以字符串 "Controller" 结尾。

控制器是在 DefaultHttpControllerSelector 信息缓存首次访问时发现的，如果没有为指定的控制器名找到唯一一个匹配的控制器，就会导致默认控制选择逻辑的错误。例如：如果没有为指定的控制器名找到一个缓存条目，框架就会向客户端返回一个 HTTP 响应 404 Not Found。另外，如果为指定的控制器名找到多个条目，那么框架会因模糊匹配抛出一个 InvalidOperationException。

假设指定的控制器名，匹配默认控制器选择程序的信息缓存中的一个条目，控制器选择程序会把相应的 HttpControllerDescriptor 返回给发起调用的 HttpControllerDispatcher。

这里还有一点要注意：控制器描述符（controller descriptor）的生命期和 HttpConfiguration 对象一样，也就是说，控制器描述符的生命期就是应用程序的生命期。

1. 支持基于属性的路由

Web API 2 增加了将路由指定为属性的功能。这些属性既可应用于控制器类，也可应用于操作方法。纯粹基于约定（convention-based）的方法使用路由参数和命名约定，进行控制器和操作的匹配，基于属性的声明式方法是对基于约定方法的补充。

使用基于属性的路由，在过程上分为两步。第一步是用 RouteAttribute 修饰控制器和 / 或操作，提供适当的路由模板值。第二步是让 Web API 将这些属性值映射到实际的路由数据，供框架在处理请求时使用。

例如，请看一个基本的问候 Web API：

```
public class GreetingController : ApiController
{
    // 默认映射到 GET /api/greeting
    public string GetGreeting()
    {
        return "Hello!";
    }
}
```

使用基于属性的路由，我们无需修改全局路由配置规则，就可以将这个控制器和操作映射到一个完全不同的 URL：

```
public class GreetingController : ApiController
{
    // 映射到 GET /services/hello
    [Route("services/hello")]
    public string GetGreeting()
    {
        return "Hello!";
    }
}
```

为了确保这个基于属性的路由正确添加到了 Web API 的路由配置中，我们必须调用 HttpConfiguration 的 MapHttpAttributeRoutes 方法：

```
config.MapHttpAttributeRoutes();
```

这种方式在配置时集成属性路由，可以减少对其他框架组件的修改。例如，因为所有解析和惯例基于属性的路由值的复杂性都在 Route Data 层进行处理，所以对 DefaultHttpControllerSelector 的影响仅限于如下部分：

```
controllerDescriptor = routeData.GetDirectRouteController();
if (controllerDescriptor != null)
{
    return controllerDescriptor;
}
```

我们从这段代码可以看出，如果通过基于属性的路由匹配可以明确得到一个控制器和 / 或操作，那么系统就会立即选择和返回相关的 HttpControllerDescriptor。否则，基于约定的控制器选择过程会试图基于类名寻找和选择一个控制器。

2. 插入定制的控制器选择程序

虽然控制器选择的默认逻辑可以满足大部分 Web API 开发场景的需求，但是在某些情况下，我们可能需要提供一个定制的选择策略。

为了重写默认的控制器选择策略，我们需要创建一个新的控制器选择程序服务，然后进行配置，使框架使用这个服务。创建一个新的控制器选择程序很简单，我们只需要编写一个实现 IHttpControllerSelector 接口的类，IHttpControllerSelector 接口定义如下所示：

```
public interface IHttpControllerSelector
{
        HttpControllerDescriptor SelectController(HttpRequestMessage request);
        IDictionary<string, HttpControllerDescriptor> GetControllerMapping();
}
```

正如其方法名所示，SelectController 方法的主要作用是，为指定的 HttpRequestMessage 选择一个控制器类型，并返回一个 HttpControllerDescriptor 对象。GetControllerMapping 方法为控制器选择程序添加了另一个功能：返回一个字典，其中包含全部控制器名以及对应的 HttpControllerDescriptor 对象。但是，目前只有 ASP.NET Web API 的 API 浏览器使用了这一功能。

我们通过 HttpConfiguration 对象的 Services 集合，配置定制的控制器选择程序，供框架使用。例如：下面一段代码展示了如何替换默认的控制器选择程序，不再在类名中寻找前缀 Controller，而是使用新的策略，让开发者指定一个定制的前缀：[1]

注 1：重写默认控制器前缀的完整代码更为复杂，不仅仅需要提供一个新的控制器选择程序。

```
const string controllerSuffix = "service";

config.Services.Replace(
    typeof(IHttpControllerSelector),
    new CustomSuffixControllerSelector(config, controllerSuffix));
```

什么是默认服务？

如果你浏览 ASP.NET Web API 的源代码，会看到在很多代码中，框架组件使用配置对象提供的通用服务，以获得实现了各种框架服务接口的对象。你可能会感到奇怪，这些接口后面的具体类型是在什么地方声明的。

在 HttpConfiguration 的构造函数中，我们可以看到如下声明：

```
Services = new DefaultServices(this);
```

我们可以看到，DefaultServices 类是 ServicesContainer 的一个实现，这个类型用于保存通用框架服务，其构造函数中设置了默认的服务对象，代码如下所示：

```
public DefaultServices(HttpConfiguration configuration)
{
    if (configuration == null)
    {
        throw Error.ArgumentNull("configuration");
    }

    _configuration = configuration;

    SetSingle<IActionValueBinder>(new DefaultActionValueBinder());
    SetSingle<IApiExplorer>(new ApiExplorer(configuration));
    SetSingle<IAssembliesResolver>(new DefaultAssembliesResolver());
    SetSingle<IBodyModelValidator>(new DefaultBodyModelValidator());
    SetSingle<IContentNegotiator>(new DefaultContentNegotiator());
    ...
}
```

这些默认的服务可以作为一个很好的起点，帮助你探索框架的默认行为，决定是否需要替换其中一个或多个默认行为，还可以更好地理解修改一个特定行为需要替换什么组件。

12.2.4　控制器激活

一旦控制器选择程序找到一个 HttpControllerDescriptor 对象，并将其返回给分发程序，分发程序就可以调用 HttpControllerDescriptor 的 CreateController 方法，获得控制器的一个实例。CreateController 转而将创建和获得控制器实例的任务委托给实现了 IHttpControllerActivator 接口的一个对象。

IHttpControllerActivator 只有一个功能，就是创建控制器实例。其定义如下：

```
public interface IHttpControllerActivator
{
    IHttpController Create(HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor,
        Type controllerType);
}
```

与控制器选择类似，控制器激活的默认逻辑由 DefaultHttpControllerActivator 类实现，并在 DefaultServices 构造函数中进行注册。

默认的控制器激活程序通过两个方法创建控制器对象。控制器激活程序首先尝试使用 ASP.NET Web API 依赖关系解析程序（dependency resolver）创建一个实例。依赖关系解析程序是 IDependencyResolver 接口的一个实现，为框架提供一个通用机制，将任务外部化，如创建对象和管理对象生命期。在 ASP.NET Web API 中，依赖关系解析程序也用于插入 IOC（Inversion-Of-Control，控制反转）容器，如 Ninject 和 Castle Windsor。依赖关系解析程序的实例通过 HttpConfiguration 对象的 DependecnyResolver 属性，向框架注册，框架会调用如 GetService(Type serviceType) 的方法，创建对象实例，而不会直接创建这些类型的实例。这种方式可以使设计的耦合更松散，提高 ASP.NET WEb API 框架本身的可扩展性，同样也适用于你自己的服务设计。

如果依赖关系解析程序没有在框架中注册，或者不能创建所需控制器类型的实例，那么默认的控制器激活程序会尝试执行指定控制器类型的无参数构造函数，创建一个实例。

控制器激活程序创建控制实例之后，会将控制器实例传回控制器分发程序，然后分发程序对控制器对象调用 ExecuteAsync 方法，将控制流转到控制器管道。调用代码如下：

```
return httpController.ExecuteAsync(controllerContext, cancellationToken);
```

和我们讨论过的大部分组件一样，控制器的 ExecuteAsync 是一个异步方法，返回一个 Task 实例。Web API 框架的组件不会因 I/O 操作阻塞线程的执行，从而提高框架自身的吞吐率，使用有限的计算资源处理更多的请求。

12.3 控制器管道

消息处理程序管道对 HTTP 请求和响应的低层处理进行了抽象，控制器管道则为开发者提供了高层编程抽象，例如：控制器、操作、模型和筛选器。对处理请求和响应中用到的这些对象进行指挥协调的，正是控制器实例自身——控制器管道因此得名。

12.3.1 ApiController

在根本上，ASP.NET Web API 控制器可以是任何实现了 IHttpController 接口的类。

`IHttpController` 接口包含一个异步执行方法，默认情况下由底层的分发程序调用：

```
public interface IHttpController
{
    Task<HttpResponseMessage> ExecuteAsync(
        HttpControllerContext controllerContext,
        CancellationToken cancellationToken);
}
```

虽然这个接口因其简单而具有极大的灵活性，但却缺乏 ASP.NET 开发者惯于使用的很多功能，例如身份验证功能、模型绑定和验证。为了提供这些功能，同时又保持接口的简单性，降低消息处理程序管道和控制器管道之间的耦合度的同时，ASP.NET Web API 团队设计了 `ApiController` 基类。`ApiController` 对核心控制器抽象进行了扩展，为派生类提供两类服务：

- 一个处理模型，其中包含筛选器、模型绑定和操作方法；
- 附加的上下文对象和帮助类，其中包含底层配置、请求消息、模型状态以及其他元素的上下文对象。

12.3.2 ApiController处理模型

`ApiController` 指挥的处理模型由几个不同阶段组成，并且和低层的消息处理程序管道一样，提供很多不同的扩展点，可以为默认的数据流提供定制逻辑。

大体上，控制器管道可以选择一个操作方法进行请求处理，将请求的属性映射到选中方法的参数，并可以执行各种筛选器类型。通过 `ApiController` 进行的请求处理大致如图 12-4 所示。

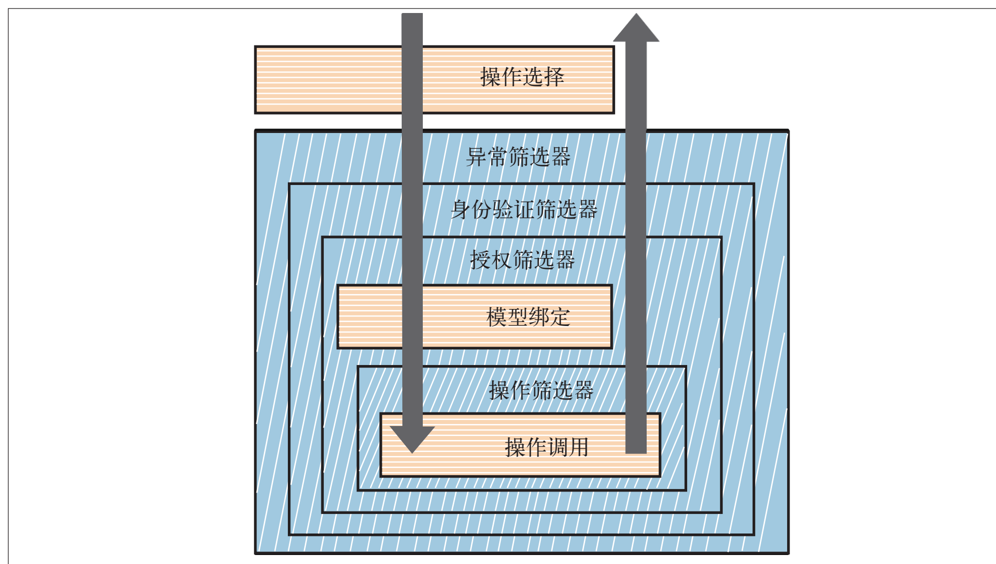


图 12-4：控制器管道

与消息处理程序管道类似，控制器管道也构建一个“俄罗斯套娃”结构，一个请求从这个结构的最外范围开始，流经一系列嵌套范围，到达最内范围的操作方法。操作方法生成一个响应，该响应从最内范围流回到最外范围。控制器管道中的范围是通过筛选器实现的，也和消息处理程序管道一样，控制器管道的所有组件都是通过任务实现异步执行。例如，管道接口 IActionFilter 就是如此。

```
public interface IActionFilter : IFilter
{
    Task<HttpResponseMessage> ExecuteActionFilterAsync(
        HttpActionContext actionContext,
        CancellationToken cancellationToken,
        Func<Task<HttpResponseMessage>> continuation);
    }
}
```

本章稍后将详细介绍筛选器。我们首先要讨论的是，基于请求特征选择控制器上正确操作的过程。

1. 操作选择

ApiController.ExecuteAsync 方法内部执行的首批操作之一是操作选择。操作选择是基于收到的 HttpRequestMessage，进行控制器方法选择的过程。与控制器选择一样，操作选择也委托给一个类型，其主要任务就是操作选择。这个类型可以是任何实现了 IHttpActionSelector 接口的类。IHttpActionSelector 的签名如下：

```
public interface IHttpActionSelector
{
    HttpActionDescriptor SelectAction(HttpControllerContext controllerContext);

    ILookup<string, HttpActionDescriptor> GetActionMapping(
        HttpControllerDescriptor controllerDescriptor);
}
```

和 HttpControllerSelector 一样，IHttpActionSelector 在技术上有两个功能：从上下文选择操作，以及提供操作映射的列表。实现了第二个功能的操作选择程序就可以由 ASP.NET Web API 的 API 浏览器功能所使用。

通过明确替换默认的操作选择程序（稍后进行讨论），或者使用一个依赖关系解析程序（通常与控制反转容器一起使用），我们可以很容易地提供一个定制的操作选择程序。例如：下面的代码使用 Ninject 控制反转容器，将默认的操作选择程序替换为名为 CustomActionSelector 的定制选择程序。

```
var kernel = new StandardKernel();
kernel.Bind<IHttpActionSelector>().To<CustomActionSelector>();

config.DependencyResolver = new NinjectResolver(kernel);
```

为了判断是否有必要提供定制的操作选择程序，你必须首先理解默认的操作选择程序实现的逻辑。Web API 提供的默认的操作选择程序是 ApiControllerActionSelector，其实现实际上是一组筛选器，这些筛选器的预期行为是从一个备选操作列表中返回一个操作。ApiControllerActionSelector 的 FindMatchingActions 方法实现了这一选择算法，如图 12-5 所示。

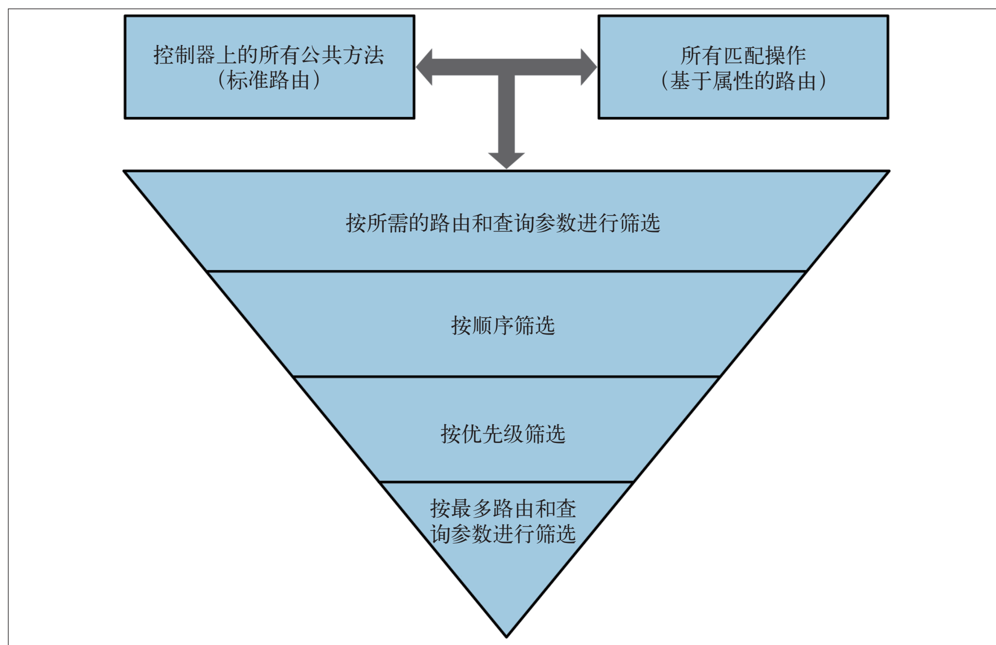


图 12-5：默认操作选择程序逻辑

在默认的操作选择逻辑中，最初的也是最关键的一点，是判断匹配的路由是标准路由（即，在 Web API 的全局配置中，通过如 MapHttpRoute 的方法声明的路由），还是使用 RouteAttribute 属性修饰操作产生的，基于属性的路由。

```
public class ValuesController
{
     [ActionName("do")]
     public string ExecuteSomething() {
         ...
     }
 }
```

如果没有找到匹配操作路由参数值的操作方法，选择程序会返回一个 HTTP 响应 404 Not Found。如果找到了匹配的操作方法，那么匹配的操作会再次进行筛选，去除收到请求相关的具体方法不适用的操作，将结果作为初始的操作备选列表返回。

如果路由数据没有明确指明操作方法，那么初始操作备选的选择逻辑会尝试从 HTTP 方法名推导出操作名。例如，对于一个 GET 请求，选择程序会寻找名字以字符串 "GET" 开头的

操作方法。

如果请求与一个基于属性的路由匹配，那么初始的备选操作列表由路由自身提供，并接着进行筛选，去除那些不适合所收到请求的 HTTP 方法的备选操作。

在建立备选操作方法的初始列表之后，默认操作选择逻辑会执行一系列的细化，将备选操作的列表缩短到只剩下一个。这些细化如下。

- 筛选掉未包含匹配路由所需的参数集的方法。
- 精简备选操作列表，只保留评估顺序值最低的操作。你可以使用 `RouteAttribute` 的 `Order` 属性，控制基于属性的路由的备用操作的顺序。在默认情况下，`Order` 设置为零，因此这个细化阶段会返回整个备选操作集合。
- 精简备选操作列表，只保留优先级最高的操作。优先级用于基于属性的路由，由 `RoutePrecedence.Compute` 函数，根据匹配的路由，计算得到。
- 将剩余的备选操作列表按照参数数目分组，从参数最多的组中，返回第一个备选操作。

到这里，备选操作列表中应该只剩下一个操作。于是，默认的操作选择程序执行一个最终检查，根据返回的备选操作数量，采取以下三种操作之一。

- 当返回的备选操作数量为 0 时：如果存在备选操作但该操作不适用于请求的 HTTP 方法，那么返回一个 HTTP 405 消息；如果没有匹配的操作，那么返回一个 HTTP 404 消息。
- 当返回的备选操作数量为 1 时，返回匹配的操作描述符，用于操作调用。
- 当返回的备选操作数量 >1 时，抛出一个 `InvalidOperationException` 异常，说明存在模糊匹配。

2. 筛选器

如图 12-4 所示，筛选器提供一个嵌套的范围集合，用于实现跨越多个控制器或者操作的功能。虽然筛选器在概念上相似，但根据其何时运行以及访问的数据类型，筛选器可以分为四类：身份验证筛选器、授权筛选器、操作筛选器和异常筛选器。图 12-6 展示了这些分类。

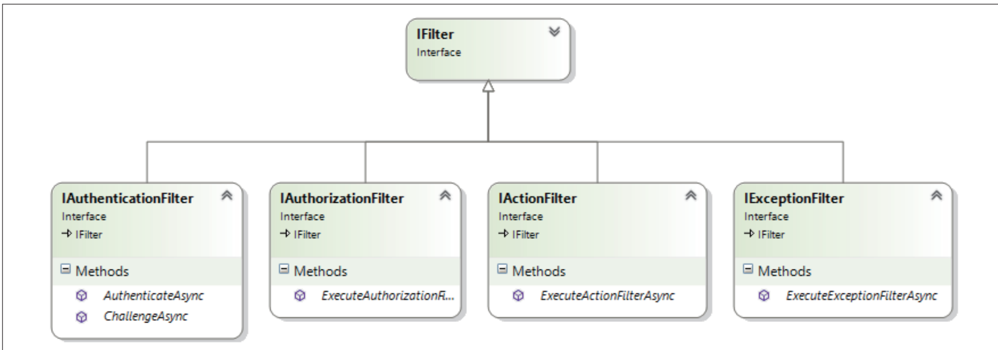


图 12-6：筛选器类

在根本上，Web API 中的筛选器可以是任何实现了 IFilter 接口的类 .IFilter 接口只包含一个方法：

```
public interface IFilter
{
    bool AllowMultiple { get; }
}
```

除了 IFilter 接口，Web API 还提供一个基类，这个基类实现了 IFilter 接口，并继承了 .NET 框架的 Attribute 类。因此，从这个基类派生的所有筛选器都可以采用两种方式添加。首先，这些筛选器可以直接添加到全局配置对象的 Filters 集合，代码如下：

```
config.Filters.Add(new CustomActionFilter());
```

此外，从基类 FilterAttribute 派生的筛选器也可以作为属性，添加到 Web API 控制器类或者操作方法，代码如下：

```
[CustomActionFilter]
public class ValuesController : ApiController
{
    ...
}
```

无论通过哪种方式加到 Web API 项目，筛选器都保存在 HttpConfiguration.Filters 集合中，这个集合将筛选器作为 IFilter 对象集合进行存储。这种泛型设计使得 HttpConfiguration 对象可以包含很多不同类型的筛选器，其中包括四种类型（身份验证筛选器、授权筛选器、操作筛选器和异常筛选器）之外的筛选器类型。我们可以创建新筛选器类型，并添加到 HttpConfiguration 中，既不会破坏应用程序，也不需要修改 HttpConfiguration。随后，这些新筛选器类型可以由定制的新控制器类型发现并运行。

ApiController 根据如下条件，对筛选器的排序和执行进行协调：

- 筛选器类型
  ApiController 按照类型对筛选器分组，将每个组作为不同的嵌套范围执行（参见图 12-4）。

- 适用处
  作为全局配置（HttpConfiguration.Filters.Add(...)）的一部分添加的筛选器首先添加到 Filters 集合中，作为类属性（ActionFilterAttribute、AuthorizationFilterAttribute 和 ExceptionFilterAttribute）或操作方法属性添加的筛选器随后添加。在通过属性添加的筛选器中，控制器类属性的筛选器首先添加，操作方法属性的筛选器随后添加。筛选器运行的顺序与添加顺序相同。因此，全局添加的筛选器首先运行，接着是通过控制器属性添加的筛选器，最后是通过操作方法属性添加的筛选器。

- *添加顺序*

  筛选器在分组之后，在组内按照添加顺序执行。

3. 身份验证筛选器

身份验证筛选器有两个功能。首先，当请求流经管道时，身份验证筛选器对请求进行检查，验证一组声明，建立调用用户的身份。如果不能从提供的声明建立用户身份，身份验证筛选器也可以用于修改响应，向用户代理提供关于建立用于身份的进一步指示。这种响应称为质询响应（challenge response）。第 15 章将详细讨论身份验证筛选器。

4. 授权筛选器

授权筛选器执行访问策略，强制用户、客户端应用程序或其他对象（指安全对象）对 Web API 提供的 HTTP 资源或资源组的访问层次。在技术上，一个授权筛选器就是实现了 `IAuthorizationFilter` 接口的类。`IAuthorizationFilter` 接口只包含一个异步运行筛选器的方法：

```
Task<HttpResponseMessage> ExecuteAuthorizationFilterAsync(
    HttpActionContext actionContext,
    CancellationToken cancellationToken,
    Func<Task<HttpResponseMessage>> continuation);
```

虽然运行授权筛选器的唯一条件是实现 `IAuthorizationFilter` 接口，但是 `IAuthorizationFilter` 接口并不是开发者最易使用的编程模型——这主要是由异步编程和 .NET 框架任务 API 的复杂性导致的。因此，Web API 提供了 `AuthorizationFilterAttribute` 类。`AuthorizationFilterAttribute` 实现了 `IAuthorizationFilter` 接口和 `ExecuteAuthorizationFilterAsync` 方法，提供如下的虚拟方法，可以由派生类重写：

```
public virtual void OnAuthorization(HttpActionContext actionContext);
```

`AuthorizationFilterAttribute` 在自己的 `ExecuteAuthorizationFilterAsync` 方法中调用了 `OnAuthorization` 方法，使我们可以用更熟悉的同步风格编写派生的授权筛选器。在调用 `OnAuthorization` 之后，基类检查 `HttpActionContext` 对象的状态，由此做出判断，是继续进行请求处理，还是返回一个新的 `Task`，其中包含新的表示授权失败的 `HttpResponseMessage`。

当你编写派生自 `AuthorizationFilterAttribute` 的定制授权筛选器时，要表示授权失败，可以将 `actionContext.Response` 设置为一个 `HttpResponseMessage` 对象，代码如下所示：

```
public class CustomAuthFilter : AuthorizationFilterAttribute
{
    public override void OnAuthorization(HttpActionContext actionContext)
    {
        actionContext.Response = actionContext.Request.CreateErrorResponse(
            HttpStatusCode.Unauthorized, "denied");
    }
}
```

当 OnAuthorize 调用结束后，AuthorizationFilterAttribute 类使用下面的代码，分析上下文状态，然后继续进行处理或者立即返回响应：

```
if (actionContext.Response != null)
{
    return TaskHelpers.FromResult(actionContext.Response);
}
else
{
    return continuation();
}
```

此外，如果 OnAuthorize 调用抛出一个异常，AuthorizationFilterAttribute 会捕获这个异常，终止处理，返回一个响应码为 500 Internal Server Error 的 HTTP 响应：

```
try
{
    OnAuthorization(actionContext);
}
catch (Exception e)
{
    return TaskHelpers.FromError<HttpResponseMessage>(e);
}
```

当你设计和实现一个新的授权筛选器时，一开始可以使用 Web API 提供的现有的 AuthorizeAttribute。AuthorizeAttribute 使用 Thread.CurrentPrincipal 获得认证用户的身份（可能还有角色成员信息），与属性构造函数中提供的策略信息进行比较。还有一个有趣的细节要注意：AuthorizeAttribute 会检查操作方法及其控制器的 AllowAnonymousAttribute 属性，如果属性存在，就成功退出。

5. 操作筛选器

操作筛选器在概念上与授权筛选器极为相似。实际上，IActionFilter 的执行方法的签名与 IAuthorizationFilter 中对应的方法签名是一样的。

```
Task<HttpResponseMessage> IActionFilter.ExecuteActionFilterAsync(
    HttpActionContext actionContext,
    CancellationToken cancellationToken,
    Func<Task<HttpResponseMessage>> continuation)
```

但是，操作筛选器与授权筛选器有几个不同之处。第一个不同之处是操作筛选器的调用时机。我们前面讨论过，筛选器按类型进行分组，不同的组在不同的时间执行。授权筛选器首先运行，然后是操作筛选器，最后是异常筛选器。

第二个不同之处是操作筛选器与其他两种筛选器最显著的不同：开发者可以在操作方法调用的前后进行请求处理。这个功能是通过 ActionFilterAttribute 类中下面两个方法提供的。

```
public virtual void OnActionExecuting(HttpActionContext actionContext);

public virtual void OnActionExecuted(
       HttpActionExecutedContext actionExecutedContext);
```

和其他类型的筛选器一样，ActionFilterAttribute 实现了 IActionFilter 接口，对直接使用 Task 的复杂度进行抽象，同时也继承了 Attribute，从而使操作筛选器可以直接应用在控制器类和操作方法上。

作为开发者，你可以简单地从 ActionFilterAttribute 进行派生，重写 OnActionExecuting 和 / 或 OnActionExecuted 方法，很容易地创建操作筛选器。例如，示例 12-3 对操作方法进行了一些基本的审计操作。

示例 12-3：审核操作方法的操作筛选器示例

```
public class AuditActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpActionContext c)
    {
      Trace.TraceInformation("Calling action {0}::{1} with {2} arguments",
         c.ControllerContext.ControllerDescriptor.ControllerName,
         c.ActionDescriptor.ActionName,
         c.ActionArguments.Count);
    }

    public override void OnActionExecuted(HttpActionExecutedContext c)
    {
      object returnVal = null;
      var oc = c.Response.Content as ObjectContent;
      if (oc != null)
         returnVal = oc.Value;

      Trace.TraceInformation("Ran action {0}::{1} with result {2}",
        c.ActionContext.ControllerContext.ControllerDescriptor.ControllerName,
        c.ActionContext.ActionDescriptor.ActionName,
        returnVal ?? string.Empty);
    }
}
```

如果操作筛选器中抛出一个异常，那么 ActionFilterAttribute 会创建一个包含错误信息的 Task<HttpResponseMessage>，从而终止管道中其他组件执行的请求处理。这种做法与前面介绍的授权筛选器的处理逻辑一致。ActionFilterAttribute 也包含额外的逻辑，当筛选器的 OnActionExecuted 方法抛出异常时，将响应消息的上下文设置为空，具体做法很简单：将 OnActionExecuted 调用封装在一个 try..catch 块中，在 catch 块中将响应设为 null。

```
try
{
   OnActionExecuted(executedContext);
   ...
   }
```

```
catch
{
   actionContext.Response = null;
   throw;
}
```

6. 异常筛选器

正如这个筛选器名字表明的，异常筛选器存在的目的是对控制器管道中抛出的异常进行定制处理。与授权筛选器和操作筛选器一样，异常筛选器是通过实现 `IExceptionFilter` 接口定义的。另外，框架还提供基类 `ExceptionFilterAttribute`，以实现 .NET 框架属性功能，并为其派生类提供一个更为简化的编程模型。

在防止服务泄露可能的敏感信息时，异常筛选器极为有用。例如，数据库异常通常包含数据库服务器或数据库模式设计的详细信息，攻击者可能利用这些信息，对你的服务器发起攻击。下面是一个异常筛选器示例，这个筛选器将异常的详细信息记录在 .NET 框架的诊断系统中，然后返回一个通用的错误响应。

```
public class CustomExceptionFilter : ExceptionFilterAttribute
{
    public override void OnException(
       HttpActionExecutedContext actionExecutedContext)
    {
       var x = actionExecutedContext.Exception;

       Trace.TraceError(x.ToString());

       var errorResponse = actionExecutedContext.Request.CreateErrorResponse(
          HttpStatusCode.InternalServerError,
          "Please contact your server administrator for more details.");

       actionExecutedContext.Response = errorResponse;
   }
}
```

如果你应用这个筛选器（作为全局筛选器，或者在控制器层或操作层使用），会用到下面的方法：

```
[CustomExceptionFilter]
public IEnumerable<string> Get()
{
    ...
    throw new Exception("Here are all of my users credit card numbers...");
}
```

这将会向客户端返回一个“删节版”的错误消息：

```
$ curl http://localhost:54841/api/values
{"Message":"Please contact your server administrator for more details."}
```

但是，如果异常筛选器抛出一个异常会怎样？更宽泛地说，如果没有异常筛选器会怎样？未处理的异常最终会在哪里捕获？答案是 HttpControllerDispatcher。如果你还记得，本章前面曾介绍过，在默认情况下 HttpControllerDispatcher 是消息处理程序管道中的最后一个组件，负责调用 Web API 控制器的 ExecuteAsync 方法。而且，HttpControllerDispatcher 将这个 ExecuteAsync 调用封装在 try..catch 块中，代码如下所示：

```
protected override async Task<HttpResponseMessage> SendAsync(
   HttpRequestMessage request, CancellationToken cancellationToken)
{
   try
   {
      return await SendAsyncCore(request, cancellationToken);
   }
   catch (HttpResponseException httpResponseException)
   {
      return httpResponseException.Response;
   }
   catch (Exception exception)
   {
      return request.CreateErrorResponse(HttpStatusCode.InternalServerError,
          exception);
    }
}
```

如你所见，这个分发程序可以返回附在 HttpResponseException 上的 HttpResponseMessage。而且，分发程序包含一段通用的异常处理代码，当捕获到未处理的异常时，异常处理代码将这个异常转换成一个 HttpResponseMessage，消息中包含异常的具体信息以及 HTTP 响应码 500 Internal Server Error。

7. 模型绑定和验证

第 13 章将重点讨论模型绑定，我们在此就不做赘述了。但是对于控制器管道而言，有一点非常重要：模型绑定恰好发生在行为筛选器处理之前，如下面的 ApiController 源代码片段所示：

```
private async Task<HttpResponseMessage> ExecuteAction(
  HttpActionBinding actionBinding, HttpActionContext actionContext,
  CancellationToken cancellationToken, IEnumerable<IActionFilter> actionFilters,
  ServicesContainer controllerServices)
{
  ...

  await actionBinding.ExecuteBindingAsync(actionContext, cancellationToken);
  _modelState = actionContext.ModelState;

  ...
}
```

这个顺序非常重要，因为这意味着行为筛选器可以使用模型状态，简化一些工作，例如，构建一个行为筛选器，在模型状态无效时自动返回 HTTP 响应 400 Bad Request。这样一

来，我们不再需要在每个 PUT 和 POST 行为方法中插入如下代码了：

```
public void Post(ModelValue v)
{
  if (!ModelState.IsValid)
  {
     var e = Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
     throw new HttpResponseException(e);
  }
}
```

相反，我们可以在一个简单的操作筛选器中进行这种模型状态检查，代码如示例 12-4 所示。

示例 12-4：模型状态验证筛选器

```
public class VerifyModelState : ActionFilterAttribute
{
   public override void OnActionExecuting(HttpActionContext actionContext)
   {
      if (!actionContext.ModelState.IsValid)
      {
         var e = actionContext.Request.CreateErrorResponse(
            HttpStatusCode.BadRequest, actionContext.ModelState);
            actionContext.Response = e;
      }
   }
}
```

8. 操作调用

控制器管道的最后一步是在控制器上调用选中的操作方法。这个任务由一个特殊的 Web API 组件操作调用程序（action invoker）完成。操作调用程序可以是实现了 IHttpActionInvoker 接口的任何类。IHttpActionInvoker 接口签名如下：

```
public interface IHttpActionInvoker
{
   Task<HttpResponseMessage> InvokeActionAsync(
      HttpActionContext actionContext, CancellationToken cancellationToken);
}
```

ApiController 从 DefaultServices 请求操作调用程序，这意味着你可以用 DefaultServices 的 Replace 方法，或者依赖注入框架和 DependencyResolver，替换操作调用程序。但是，Web API 框架提供的默认实现 ApiControllerActionInvoker，应该能够满足大部分的需求。

默认调用程序执行两个主要功能，如下面的代码所示：

```
object actionResult = await actionDescriptor.ExecuteAsync(controllerContext,
   actionContext.ActionArguments,
   cancellationToken);

return actionDescriptor.ResultConverter.Convert(controllerContext, actionResult);
```

正如你所预期的，默认调用程序的第一个功能是调用选中的操作方法。第二个功能是将操作方法调用的结果转换成一个 HttpResponseMessage。为了实现第二个功能，调用程序使用了一个特殊的对象，操作方法转换器（action result converter）。操作方法转换器实现了 IActionResultConverter 接口，这个接口只包含一个方法，该方法参数为一些上下文数据，返回一个 HttpResponseMessage。目前，Web API 包含三种操作结果转换器：

- ResponseMssageResultConverter

  当操作方法直接返回 HttpResponseMessage 时使用，直接传回响应消息。

- ValueResultConverter<T>

  当操作方法返回一个 .NET 框架标准类型时使用，使用相关的 HttpRequestMethod 的 CreateResponse<T> 方法，创建一个 HttpResponseMessage。

- VoidResultConverter

  当操作方法返回值为空时使用，创建一个新的 HttpResponseMessage，状态码为 204 No Content。

操作方法的返回值转换为 HttpResponseMessage 之后，这个消息就可以开始反向流出控制器和消息处理程序管道，然后传送给客户端。

12.4 小结

本章，我们深入探索了 ASP.NET Web API 中进行请求处理的两个管道：低层的消息处理程序管道和控制器管道。这两种管道各有利弊。例如，消息处理程序管道在请求处理的早期执行，因此适合终止执行代价较高的代码路径，但随之而来的代价是，你必须在 HttpRequestMessage 和 HttpResponseMessage 层次工作。另一方面，控制器管道为其组件提供了较高层次的编程模型对象，例如描述控制器、操作方法和相关属性的对象。这两种管道都提供一整套默认的组件，以及灵活的模型，可以使用 HttpConfiguration 或定制的 DependencyResolver 进行扩展。

在下一章，我们将更深入地讨论组成消息处理器管道的核心构件，其中有消息处理程序自身，也有 HTTP 原始类型 HttpRequestMessage 和 HttpResponseMessage。

第 13 章 格式化程序和模型绑定

任何人都能写出计算机可以理解的代码，但只有好的程序员才能写出人工可读的代码。

之前我们讨论过，使用媒体类型及其语义，表示系统中领域空间的概念。一旦涉及具体实现，这些抽象概念就必须翻译为程序员能理解的术语。对于 ASP.NET Web API，这些抽象概念的最终表示就是对象，或者更准确地说，模型。模型代表了一种抽象层次，开发者使用模型将对象映射到媒体类型表示，或者 HTTP 消息的其他不同部分。

ASP.NET Web API 中的模型绑定（model binding）基础结构为我们提供了进行这些映射必需的运行时服务。有了模型绑定，开发者可以专注于 Web API 的实现细节，将所有的序列化问题留给框架处理。使用这种架构有一个显而易见的好处，开发者可以使用单个抽象层次，即模型，根据 Web API 的不同使用者的需求，支持各种媒体类型。例如，在问题跟踪应用程序中，我们使用表示问题的单个模型类，这个类可以由框架转换成不同的媒体类型，如 JSON 或 XML。

本章将探索模型绑定，详细介绍不同的运行时组件，以及框架提供的扩展性挂钩（extensibility hook），用于定制或添加新的模型绑定功能。

13.1 ASP.NET Web API中模型的重要性

一般来说，解决单个问题的控制器操作，从长远来看比较容易测试、扩展和维护。将消息表示转换成模型对象，就是应该首先从操作实现中移出的问题之一。请看示例 13-1，代码中混合了序列化处理和 Web API 操作的实现。

示例 13-1：包含序列化的操作

```
public HttpResponseMessage Post(HttpRequestMessage request) // <1>
{
  int id = int.Parse(request.RequestUri.ParseQueryString().Get("id")); // <2>

  var values = request.Content.ReadAsFormDataAsync().Result // <3>

  var issue = new Issue
  {
     Id = id,
     Name = values["name"],
     Description = values["description"]
  };

  // 处理已构建的问题
}
```

这段代码有几个明显的问题，如下。

- 控制器方法的签名非常通用，如果不查看实现细节，我们很难推测这个方法的用途，也无法使用不同的参数重载 `Post` 方法，以支持多个场景。
- 代码没有检查查询字符串中的参数是否存在，或者是否能转换成整数。
- 代码将实现绑定到单个媒体类型（`application/form-url-encoded`），而且执行线程无法同步读取正文内容。对于最后一点，直接调用异步任务的 `Result` 属性而不检查其是否完成，可能会阻止执行线程返回线程池处理新的请求，因此不是好的做法。

我们可以只使用一个模型类，很容易地重写这个操作方法，避免以上问题，如代码 13-2 所示。

示例 13-2：使用模型绑定的操作

```
public void Post(Issue issue) // <1>
{
  // 处理已构建的问题
}
```

如你所见，所有的序列化处理都从实现中消失了，操作中只保留了关键的代码。Web API 框架中的模型绑定基础结构会在执行这个操作时处理其他的问题。

13.2　模型绑定如何工作

模型绑定基础结构的最核心部分，是一个称为 `HttpParameterBinding` 的组件，这个组件知道如何从一个请求消息得到参数值（参见示例 13-3）。每个 `HttpParameterBinding` 实例都和一个参数联系在一起，这个参数在 Web API 执行操作时用 `HttpConfiguration` 进行定义。`HttpParameterBinding` 实例如何联系到一个参数，是由另一个配置类 `HttpParameterDescriptor` 决定的，这个配置类包含了描述参数的元数据，即参数名、类型，或者模型绑定基础结构可以用来选择 `HttpParameterBinding` 的其他任何属性。

示例 13-3：`HttpParameterBinding` 操作

```
public abstract class HttpParameterBinding
{
  protected HttpParameterBinding(HttpParameterDescriptor descriptor);

  public abstract Task ExecuteBindingAsync(ModelMetadataProvider metadataProvider,
  HttpActionContext actionContext, CancellationToken cancellationToken); // <1>
}
```

示例 13-3 展示了 `HttpParameterBinding` 的基本结构，其中关键方法是 `ExecuteBindingAsync`，`HttpParameterBinding` 中每个实现都必须实现这个方法，以执行参数绑定。

和 ASP.NET Web API 中的很多运行时组件一样，`HttpParameterBinding` 也为其核心方法 `ExecuteBindingAsync` 提供异步的方法签名。如果你有一个实现，它不依赖于从当前请求消息获取的值，而是执行一些 I/O 操作，例如查询数据库或者读取文件，那么这个异步方法就会很有用。示例 13-4 展示了 `HttpParameterBinding` 的一个基本实现，从执行线程的区域性集合，进行操作参数中 `CultureInfo` 类型参数的绑定。

示例 13-4：`HttpParameterBinding` 实现

```
public class CultureParameterBinding : HttpParameterBinding
{
  public CultureParameterBinding(HttpParameterDescriptor descriptor) // <1>
    : base(descriptor)
  {
  }

  public override System.Threading.Tasks.Task
  ExecuteBindingAsync(System.Web.Http.Metadata.ModelMetadataProvider
  metadataProvider, HttpActionContext   actionContext,
  System.Threading.CancellationToken cancellationToken)
  {
    CultureInfo culture = Thread.CurrentThread.CurrentCulture; // <2>
    SetValue(actionContext, culture); // <3>

    var tsc = new TaskCompletionSource<object>(); // <4>
    tsc.SetResult(null);
    return tsc.Task;
  }
}
```

`HttpParameterBinding` 的实例创建时使用一个描述符。我们的实现忽略了这个参数，但是别的实现可能会用到这个参数的一些信息 <1>。`ExecuteBindingAsync` 方法从当前线程中获得 `CultureInfo` 实例 <2>，并调用基类中的 `SetValue` 方法，使用获得的 `CultureInfo` 实例设置绑定 <3>。`ExecuteBindingAsync` 方法在最后创建了一个 `TaskCompletionSource`，用于返回已经同步完成的新任务 <4>。在这个方法的异步版本中，`SetValue` 可能会作为返回任务的一部分调用。

我们现在可以使用这个 CultureParameterBinding，将一个 CultureInfo 实例直接作为操作方法的参数传入，如示例 13-5 所示。

示例 13-5：以 CultureInfo 实例为参数的 Web API 操作

```
public class HomeController : ApiController
{
  [HttpGet]
  public HttpResponseMessage BindCulture(CultureInfo culture)
  {
    return Request.CreateResponse(System.Net.HttpStatusCode.Accepted,
        String.Format("BindCulture with name {0}.", culture.Name));
  }
```

你现在了解了 HttpParameterBinding 是什么，但是我们还没有讨论，当操作执行时，框架如何配置和选择 HttpParameterBinding。HttpParameterBinding 的选择是在框架中可用的诸多可插入服务之一，System.Web.Http.ModelBinding.IActionValueBinder 中完成的，IActionValueBinder 的默认实现是 System.Web.Http.ModelBinding.DefaultActionValueBinder。IActionValueBinder 负责返回一个 HttpActionBinding 实例，这个实例主要包含与指定控制器操作相关的 HttpParameterBinding 实例集合，这个集合可以进行缓存，供多个请求使用。

```
public interface IActionValueBinder
{
  HttpActionBinding GetBinding(HttpActionDescriptor actionDescriptor);
}
```

DefaultActionValueBinder 的内建实现使用了反射机制，构建一个 HttpParameter Descriptor 列表，用于之后的配置查询和 HttpParamterBinding 实例的选择（参见图 13-1）。

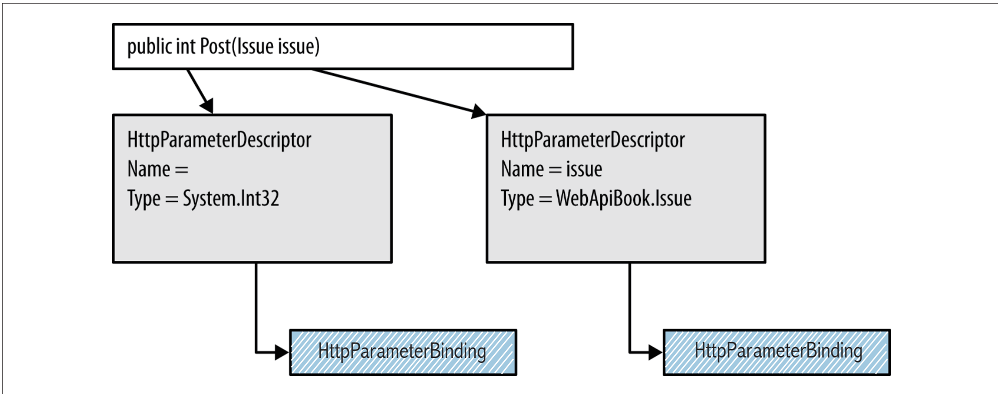


图 13-1：HttpParameterBinding 选择

目前，DefaultActionValueBinder 支持两种不同的方式，判断与一个操作关联的是哪个 HttpParameterBinding。第一种方式通过使用 HttpConfiguration 对象的 Parameter

BindingRules 属性建立关联，这个属性提供了一组规则，用于为一个指定的 HttpParameterDescriptor 选择绑定实例。这些规则的形式是一个委托 Func<HttpPara meterDescriptor,HttpParameterBinding>，这个委托以描述符为参数，返回一个绑定实例。这意味着，你可以提供一个方法回调，或者一个 Lamda 表达式，用于绑定的解析。对于 CultureParameterBinding，我们需要定义一个规则，为与类型 System.Globalization.CultureInfo 相关联的 HttpParameterDescriptor 返回 CultureParameterBinding，代码如示例 13-6 所示。

示例 13-6：使用规则配置 HttpParameterBinding

```
config.ParameterBindingRules.Insert(0, (descriptor) => // <1>
{
    if (descriptor.ParameterType == typeof(System.Globalization.CultureInfo)) // <2>
      return new CultureParameterBinding(descriptor);

    return null;
});
```

这个插入的新规则使用一个 Lambda 表达式 <1>，检查描述符的 ParameterType 属性，只有当类型为 System.Globalization.CultureInfo 时才返回我们的绑定 <2>。

第二种机制是声明性的，用到一个属性 ParameterBindingAttribute。使用这种机制，我们需要派生这个属性（详见下一节）。

如果没有找到映射规则或者 ParameterBindingAttribute，系统会使用一种默认策略，将简单类型绑定到 URI 片段或查询字符串变量，将复杂类型绑定到请求正文。

13.3 内建的模型绑定器

ASP.NET Web API 框架自带几个内建的绑定器实现，但是对于开发者来说，只有三个绑定器值得特别注意：ModelBindingParameterBinder、FormatterParameterBinder 和 HttpRequestParameterBinding。这个三个绑定器以完全不同的方式，实现了消息各部分到模型的绑定。第一个绑定器 ModelBindingParameterBinder，借用了 ASP.NET MVC 的一种方式，将消息的不同部分像乐高玩具块一样进行组合，形成模型。第二个 FormatterParameterBinder，依赖于格式化程序，理解给定媒体类型的全部语义和格式，并知道如何应用这些语义进行模型的序列化或者反序列化。格式化程序代表了内容协商的一个关键部分，是首选的绑定方式。最后，第三个 HttpRequestParameterBinding，用于支持那些直接在方法签名中使用 HttpRequestMessage 或 HttpResponseMessage 实例的操作。

13.3.1 ModelBindingParameterBinder

ModelBindingParameterBinder 的实现重用了 ASP.NET MVC 中进行模型绑定的方法。在这种方法中，值提供程序从 HTTP 消息的不同部分获得数据，由模型绑定器把这些部分组合成为模型。

这种实现主要关注简单的键 / 值对的绑定，例如：HTTP 标头中的键 / 值、URL 片段、查询字符串或者用 `application/form-url-encoded`（用于对 HTTP 表单进行编码的媒体类型）编码的正文。这些值通常都是消息中的字符串，并可以转换成基本类型。模型绑定器并不知道媒体类型的具体信息，也不知道如何解释媒体类型，这些是*格式化程序*（formatter）的工作。我们将在下一节详细讨论格式化程序。

ASP.NET Web API 框架自带几个内建的模型绑定器，可以将 HTTP 消息中的不同小“部件”组合成相当复杂的模型。更准确地说，这些实现也负责在给模型“补水”之前，将字符串转换为简单数据类型，如 `Timespan`、`Int`、`Guid`、`Decimal` 或带有类型转换器的其他类型。这种内建的模型绑定器实现有：`ArrayModelBinder` 和 `TypeConverterModelBinder`，二者都位于 `System.Web.Http.Model.Binding.Binders` 命名空间中。值得一提的是，模型绑定器大多数用于“恢复”简单类型，或者为构建更复杂的类型提供组件。这些内建实现通常覆盖了最常见的场景，因此，如果你要从头编写一个新的模型绑定器，最好还是三思而后行。

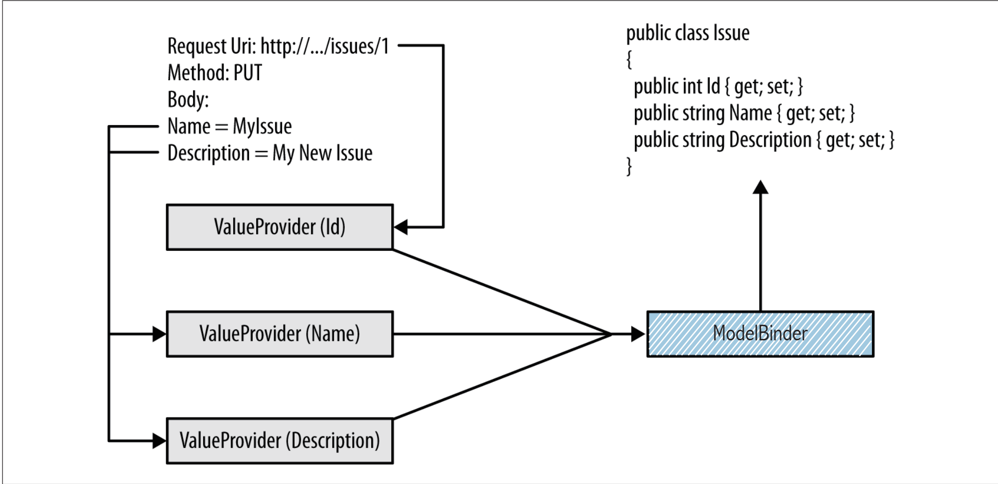


图 13-2：模型绑定工作示意图

在图 13-2 中，已配置的值提供程序首先将消息分解为片段，以获得不同的值，例如：查询字符串中的问题 ID，以及消息正文中的其他字段。这些字段使用 URL 表单编码媒体类型，通过 HTTP `PUT` 提交。选中的模型绑定器与值提供程序密切合作，获得初始化一个新的 `Issue` 实例所需的数据。

13.3.2 值提供程序

值提供程序将 HTTP 消息不同部分的值聚合在一起，提供一个不变的使用接口，从而形成一个简单的抽象层，将模型绑定器与消息细节隔离。

在其核心层，每个值提供程序都实现了 `System.Web.Http.ValueProviders.IValueProvider`

接口。接口定义见示例 13-7。

示例 13-7：`IValueProvider` 接口定义

```
public interface IValueProvider
{
  bool ContainsPrefix(string prefix); // <1>
  ValueProviderResult GetValue(string key); // <2>
}
```

第一个方法 `ContainsPrefix` <1> 返回一个布尔值，说明这个值提供程序的实现是否能为以参数 `prefix` 为前缀的键提供值，这个前缀通常代表需要反序列化的模型中的一个属性名。

第二个方法 `GetValue` <2> 可能是最重要的，这个方法在 HTTP 消息中搜索传入的键，返回相关的值。`GetValue` 不直接返回原始字符串，而是返回一个 `ValueProviderResult` 实例，这个实例提供方法，以获得原始字符串或者转换为指定类型的值。

你也许希望创建一个新的值提供程序，或者继承一个已有的值提供程序，以解决新的用例，例如，在请求消息中使用特定的命名约定搜索值，或在其他地方（如定制 cookie）进行搜索。

示例 13-8 展示了值提供程序的一个基本实现，使用一个供应商前缀 `X-`，进行标头搜索。

示例 13-8：`IValueProvider` 实现

```
public class HeaderValueProvider : IValueProvider
{
  public const string HeaderPrefix = "X-";

  private HttpControllerContext context;

  public HeaderValueProvider(HttpControllerContext context) // <1>
  {
    this.context = context;
  }

  public bool ContainsPrefix(string prefix)
  {
    var contains = context.Request
      .Headers
      .Any(h => h.Key.Contains(HeaderPrefix + prefix)); // <2>
    return contains;
  }

  public ValueProviderResult GetValue(string key)
  {
    if (!context.Request.Headers.Any(h => h.Key == HeaderPrefix + key))
      return null;

    var value = context.Request
      .Headers
      .GetValues(HeaderPrefix + key).First(); // <3>
```

```
        var stringValue = (value is string) ? (string)value : value.ToString(); // <4>

        return new ValueProviderResult(value, stringValue,
          CultureInfo.CurrentCulture); // <5>
      }
    }
```

HeaderValueProvider 的构造函数参数为一个 HttpControllerContext 实例，这个实例提供了代码执行的上下文以及请求消息 <1>。对于 HTTP 请求标头中任何以 X-prefix 为前缀的键，ContainsPrefix 方法返回 true，GetValue 方法返回键值 <3>。GetValue 方法的返回值为 ValueProviderResult 实例 <4>，其中包含值的原始字符串 <5>。

IValueProvider 实现可以在运行时通过一个值提供程序工厂注入，值提供程序工厂是抽象类 System.Web.Http.ValueProvider.ValueProviderFactory 的派生类，重写了 GetValueProvider 方法，以返回 IValueProvider 实现的实例。示例 13-9 是对应于 HeaderValueProvider 的值提供程序工厂。

示例 13-9：ValueProviderFactory 实现

```
public class HeaderValueProviderFactory : ValueProviderFactory
{
  public override IValueProvider GetValueProvider(HttpActionContext actionContext)
  {
    return new HeaderValueProvider(actionContext.ControllerContext); // <1>
  }
}
```

HeaderValueProviderFactory 以当前的 HttpActionContext 为构造函数参数，实例化了一个新的 HeaderValueProvider <1>。我们可以使用全局的依赖关系解析程序，在 HttpConfiguration 对象中注册这个工厂类（参见示例 13-10）。

示例 13-10：在 Web 托管配置中注入 HeaderValueProviderFactory

```
public static void RegisterValueProvider(HttpConfiguration config)
{
  var valueProviderFactories = config.ServiceResolver
     .GetValueProviderFactories().ToList();

  valueProviderFactories.Insert(0, new HeaderValueProviderFactory()); // <1>

  config.ServiceResolver.SetServices(typeof(System.Web.Http.ValueProviders
     .ValueProviderFactory),
     valueProviderFactories.ToArray()); // <2>
}
```

HeaderValueProviderFactory 被加入到已有的值提供程序工厂列表的第一位 <1>，因此在需要提供数据时具有优先权，之后作为服务注入依赖关系解析程序中 <2>。

ASP.NET Web API 框架提供的最重要的值提供程序是 System.Web.Http.ValueProviders.

Providers.QueryStringValueProvider 和 System.Web.Http.ValueProviders.Providers.RouteDataValueProvider，对应的工厂类分别为 System.Web.Http.ValueProviders.Providers.QueryStringValueProviderFactory 和 System.Web.Http.ValueProviders.Providers.RouteDataValueProvider。QueryStringValueProvider 解析和提供查询字符串中的值，而 RouteDataValueProvider 负责从路由参数（即路由配置中路由层定义的参数）中获取值。

13.3.3　模型绑定器

模型绑定器负责协调从已配置的值提供程序请求的不同数据，“组装”形成一个新的模型实例的所有操作。模型绑定器实现了 System.Web.Http.ModelBinding.IModelBinder 接口，这个接口只包含一个方法 BindModel，所有的关键代码都在这个方法中（参见示例 13-11）。

示例 13-11：IModelBinder 接口

```
public interface IModelBinder
{
  bool BindModel(HttpActionContext actionContext, ModelBindingContext
  bindingContext); // <1>
}
```

BindModel 方法有两个参数 <1>：一个是 HttpActionContext 实例，其中包含了当前执行线程的特定信息；另一个是 ModelBindingContent 实例，表示模型绑定过程的上下文。BindModel 方法返回一个布尔值，表明绑定器是否成功地组装了一个新的模型实例。绑定上下文提供两个重要的属性：ModelState 和 ModelMetadata。ModelState 是一个属性包类，供模型绑定器存储绑定模型过程的结果，或者过程中可能发生的任何错误。ModelMetadata 提供了对已发现的模型元数据的访问，例如：可用属性，或者执行数据验证的任何组件模型属性。IModelBinder 接口看似非常简单，其中却隐藏了复杂的逻辑，实现模型绑定器，并在运行时提供正确的行为。因此，我们接下来要详细描述，一个 IModelBinder 实现要执行的所有步骤。

(1) 代码使用从 BindModel 的绑定上下文参数中得到的值提供程序，尝试获得组装新模型需要的所有值。虽然绑定上下文只提供对一个值提供程序的访问，但这个实例通常代表一个内建的值提供程序 CompositeValueProvider，这个值提供程序实现了 IValueProvider 接口，而其内部将方法调用委托给所有已配置的值提供程序。
(2) 代码创建一个模型，并使用从值提供程序获得所有值进行初始化。如果模型初始化过程中发生了错误，那么代码将异常保存在绑定上下文的 ModelState 属性中。
(3) 代码将模型保存在绑定上下文中。

示例 13-12 中的模型绑定器实现，创建了本章之前讨论过的 Issue 模型类。

示例 13-12：`IssueModelBinder` 实现

```
public class IssueModelBinder : IModelBinder
{
  public bool BindModel(HttpActionContext actionContext, ModelBindingContext
  bindingContext)
  {
    var model = (Issue)bindingContext.Model ?? new Issue();

    var hasPrefix = bindingContext.ValueProvider
      .ContainsPrefix(bindingContext.ModelName);

    var searchPrefix = (hasPrefix) ? bindingContext.ModelName + "." : "";

    int id = 0;
    if(int.TryParse(GetValue(bindingContext, searchPrefix, "Id"), out id)
    {
      model.Id = id; // <1>
    }

    model.Name = GetValue(bindingContext, searchPrefix, "Name"); // <2>
    model.Description = GetValue(bindingContext, searchPrefix, "Description");// <3>

    bindingContext.Model = model;

    return true;
  }

  private string GetValue(ModelBindingContext context, string prefix, string key)
  {
    var result = context.ValueProvider.GetValue(prefix + key); // <4>
    return result == null ? null : result.AttemptedValue;
  }
}
```

这个实现使用了绑定上下文中的值提供程序 <1> 进行数据请求，随后在 <2>、<3> 和 <4> 中将这些数据绑定到模型的属性。你在实际的应用程序中可能不会这么做，但是这个实现简单演示了 `IModelBinder` 实现的大致内容。

在运行时，模型绑定器的实现最终通过模型绑定器提供程序，进行配置和注入。模型绑定器提供程序是一个工厂类，继承了基类 `System.Web.Http.ModelBinding.ModelBinderProvider`，并实现 `GetBinder` 方法，返回一个新的模型绑定程序实例（参见示例 13-13）。

示例 13-13：返回 `IssueModelBinder` 实例的 `ModelBinderImplementation` 实现

```
public class IssueModelBinderProvider : ModelBinderProvider
{
  public override IModelBinder GetBinder(HttpActionContext actionContext,
  ModelBindingContext bindingContext)
  {
    return new IssueModelBinder();
  }
}
```

你可以使用 HttpConfiguration 对象中的依赖关系解析程序注册这个提供程序，或者给模型类添加一个 System.Web.Http.ModelBinding.ModelBinderAttribute 属性（参见示例 13-14 和示例 13-15）。

示例 13-14：带有 ModelBinderAttribute 属性的模型类

```
[ModelBinder(typeof(IssueModelBinderProvider))]
public class Issue
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Description { get; set; }
}
```

示例 13-15：带有 ModelBinderAttribute 属性的参数

```
public void Post([ModelBinder(typeof(IssueModelBinderProvider))]Issue issue)
{
}
```

关于 ModelBinderAttribute 有一个有趣的事实：ModelBinderAttribute 派生自之前讨论过的 ParameterBindingAttribute 属性。ParameterBindingAttribute 用于通过声明将一个 HttpParamterBinding 实例附到一个参数上。ModelBinderAttribute 初始化一个新的 ModelBindingParameterBinder 实例，该实例内部使用 ModelBinderProvider 为参数（在我们的示例中是 IssueModelBinderProvider）。

13.3.4　只对URI进行模型绑定

ASP.NET Web API 框架还提供另外一个属性 FromUriAttribute，这个属性派生自 ModelBinderAttribute，强制运行时只对 URL 中可用的数据执行绑定，可以用于将 URL 中找到的值绑定到模型类的属性，因为在默认情况下框架只将 URL 中的数据绑定到简单类型。

示例 13-16 展示了查询字符串变量 Lang 和 Filter 如何自动映射到 IssueFilters 模型上的同名属性。

示例 13-16：查询字符串变量的模型绑定

```
// url 示例：http://../Issues?Lang=en&Filter=2345

public class IssueFilters
{
  public string Lang { get; set; }
  public string Filter { get; set; }
}

public IEnumerable<Issue> Get([FromUri]IssueFilters)
{
  // 操作实现代码
}
```

13.3.5 FormatterParameterBinder实现

ASP.NET Web API 中引入了格式化程序，以更好地支持使用媒体类型时的内容协商。FormatterParameterBinder 实现依赖于格式化程序。在 ASP.NET MVC 中，只有 HTML（text/html）和 JSON（application/json）是第一类媒体类型，在整个栈中得到支持。而且，ASP.NET MVC 中没有支持内容协商的一致模型。你可以提供定制的 AxtionResult 实现，为响应消息支持不同的媒体类型，但是框架没有明确规定如何引入和处理新的媒体类型。开发者通常会利用模型绑定基础结构，使用新的模型绑定器或者值提供程序，来解决这个问题。

幸好，ASP.NET Web API 引入了格式化程序的概念，解决了 ASP.NET MVC 中的不一致问题。现在，格式化程序提供单一的入口点，使用媒体类型表达的格式，进行模型的序列化和反序列化，从而统一了序列化问题的处理。内容协商算法将决定对于给定的消息使用哪种格式化程序。

每个格式化程序都派生自基类 MediaTypeFormatter（参见示例 13-17），并重写 CanReadType 和 ReadFromStreamAsync 方法以支持反序列化，对 CanWriteType 和 WriteToStreamAsync 方法进行重写，以根据媒体类型的语义和格式进行模型的序列化。

示例 13-17：MediaTypeFormatter 类定义

```
public abstract class MediaTypeFormatter
{
  public Collection<Encoding> SupportedEncodings { get; }

  public Collection<MediaTypeHeaderValue> SupportedMediaTypes { get; }

  public Collection<MediaTypeMapping> MediaTypeMappings { get; }

  public abstract bool CanReadType(Type type);

  public abstract bool CanWriteType(Type type);

  public virtual Task<object> ReadFromStreamAsync(Type type, Stream readStream,
    HttpContent content, IFormatterLogger formatterLogger);

  public virtual Task WriteToStreamAsync(Type type, object value,
    Stream writeStream, HttpContent content, TransportContext transportContext);
}
```

下面罗列了 MediaFormatter 类的主要特征。

- CanReadType 和 CanWriteType 方法的参数是一个类型，必须返回一个值，说明这个格式化程序是否能够从代表消息正文的流中读取，或者向流中写入该类型的一个对象。例如，一个格式化程序可能知道如何写入一个类型，但是不知道如何从流中读取该类型。

- SupportedMediaTypes 集合包含了这个格式化程序支持的媒体类型列表（例如：text/html）。这个列表通常在格式化程序类的构造函数中初始化。运行时必须在内容协商握手中，基于 CanReadType 或 CanWriteType 方法的返回值，以及格式化程序支持的媒体类型，决定使用哪个格式化程序。值得一提的是，当 Content-Type 标头设为 multipart 时，请求消息可以混合使用不同的媒体类型，由消息的每个部分定义自己的媒体类型。运行时可以为消息中存在的所有媒体类型选择一个或多个格式化程序，处理混合媒体类型的消息。
- 对于读写操作，MediaTypeFormatter 遵循任务并行库（Task Parallel Library，TPL）编程模型。大部分的实现只涉及序列化操作，因此仍将同步执行。
- 格式化程序可以使用 MediaTypeMappings 集合，定义如何寻找与一个请求消息相关的媒体类型（如查询字符串和 HTTP 标头）。例如，一个客户端应用程序可能在查询字符串中发送预期的响应媒体格式。

ASP.NET Web API 框架提供一组直接可用的格式化程序，可以处理大部分的常用媒体类型，例如：表单编码数据（FormUrlEncodedMediaTypeFormatter）、JSON（JsonMediaTypeFormatter）或者 XML（XmlMediaTypeFormatter）。对于其他的媒体类型，你需要实现自己的格式化程序，或者使用开源社区提供的诸多实现中的一个。

JsonMediaTypeFormatter 和 XmlMediaTypeFormatter

值得一提的是，目前 JsonMediaTypeFormatter 在内部使用 Json.NET 库，进行 JSON 有效载荷的序列化 / 反序列化，XmlMediaTypeFormatter 使用 .NET 框架中的 DataContractSerializer 或 XmlSerializer 类。XmlMediaTypeFormatter 提供一个布尔型的属性 UserXmlSerializer，用于设置是否使用 XmlSerializer 类，UserXmlSerializer 的默认值为 false。你可以扩展这些类，使用自己偏好的库，进行 XML 或 JSON 的序列化。

现在，我们要讨论 MediaTypeFormatter 的一个实现，将一个模型进行序列化，成为 RSS 或 ATOM 源的一部分（参见示例 13-18）。

示例 13-18：MediaTypeFormatter 实现

```
public class SyndicationMediaTypeFormatter : MediaTypeFormatter
{
  public const string Atom = "application/atom+xml";
  public const string Rss = "application/rss+xml";

  public SyndicationMediaTypeFormatter()
    : base()
  {
    this.SupportedMediaTypes.Add(new MediaTypeHeaderValue(Atom)); // <1>
    this.SupportedMediaTypes.Add(new MediaTypeHeaderValue(Rss));
  }
```

```
public override bool CanReadType(Type type)
{
  return false;
}

public override bool CanWriteType(Type type)
{
  return true; // <2>
}

public override Task WriteToStreamAsync(Type type, object value, Stream
writeStream, HttpContent content, TransportContext transportContext) // <3>
{
  var tsc = new TaskCompletionSource<AsyncVoid>(); // <4>
  tsc.SetResult(default(AsyncVoid));

  var items = new List<SyndicationItem>();

  if (value is IEnumerable)
  {
    foreach (var model in (IEnumerable)value)
    {
      var item = MapToItem(model);
      items.Add(item);
    }
  }
  else
  {
    var item = MapToItem(value);
    items.Add(item);
  }

  var feed = new SyndicationFeed(items);

  SyndicationFeedFormatter formatter = null;
  if (content.Headers.ContentType.MediaType == Atom)
  {
    formatter = new Atom10FeedFormatter(feed);
  }
  else if (content.Headers.ContentType.MediaType == Rss)
  {
    formatter = new Rss20FeedFormatter(feed);
  }
  else
  {
    throw new Exception("Not supported media type");
  }

  using (var writer = XmlWriter.Create(writeStream))
  {
    formatter.WriteTo(writer);

    writer.Flush();
    writer.Close();
```

```
    }

    return tsc.Task; // <5>
  }

  protected SyndicationItem MapToItem(object model) // <6>
  {
    var item = new SyndicationItem();
    item.ElementExtensions.Add(model);

    return item;
  }

  private struct AsyncVoid
  {
  }
}
```

这个实现只知道如何根据 Atom 和 RSS 媒体类型定义进行模型的序列化，因此在构造函数中明确声明了这两个类型 <1>。`SyndicationMediaTypeFormatter` 的 `CanWrite` 方法返回 `true`，说明这个实现只支持写操作 <2>。

`WriteToStreamAsync` 方法的实现 <3> 主要依赖 WCF Web 编程模型中的联合（syndication）类，将模型序列化为 Atom 或 RSS 源。这个编程模型提供了构建一个联合源（syndication feed）和所有相关条目的类，还提供格式化程序类，将这些源和条目转化为常见的联合源格式，例如：Atom 或 RSS。

我们前面提到，`WriteToStreamAsync` 和 `ReadFromStreamAsync` 方法使用新的任务并行库进行异步操作。这两个方法都返回一个 `Task` 实例，其中封装了异步操作。但是，在大多数时候，序列化操作是安全的，可以同步完成。实际上，.NET 框架中的很多序列化程序类都执行同步操作。最简单的做法是，使用 `Task.Factory.StartNew` 方法，为所有的序列化工作创建一个新任务，但是这种做法会产生一些副作用。在调用 `StartNew` 方法之后，新任务计划执行，可能会产生一个线程上下文切换，对性能产生影响。在这种情况下，诀窍是使用一个 `TaskCompletionSource`<4>。`TaskCompletionSource` 标记为 complete，因而此后的所有工作都会同步完成，返回与 `TaskCompletingSource` 关联的最终任务 <5>。`MapToItem` 方法 <6> 只是简单地将模型实例用做一个联合源项（syndication item）的内容。

同步格式化程序

大多数的格式化程序都是同步的，使用一个 `TaskCompletionSource` 实例返回完成的任务。但是，如果你希望使实现更为简单，有一个基类 `BuffteredMediaFormatter` 可以在内部完成所有这些工作。这个基类提供两个可以在实现中重写的方法：`SaveToStream` 和 `ReadFromStream`，这两个方法分别是 `SaveToStreamAsync` 和 `ReadFromStreamAsync` 的同步版本。

在格式化程序中，我们可能希望支持的另一个功能是，使用 `Accept` 标头之外的信息源进行内容协商。如今，在没有正确实现 HTTP 客户栈的客户端（例如一些旧版本移动设备的浏览器）中，我们经常会需要这种功能。在这种情况下，一个客户端可能希望在查询字符串中提供可接受的媒体类型，例如：`http://.../Issues?format=atom`。`MediaTypeFormatter` 通过 `MediaMappings` 属性提供对这种场景的支持，`MediaMappings` 属性是 `MediaMapping` 实例的集合，`MediaMapping` 表明了可以找到媒体类型的位置，例如：查询字符串、标头或 URI 段。ASP.NET Web API 框架提供了抽象类 `MediaMapping` 的几个具体实现，以解决最为常见的情况。以下是这些映射类的简单介绍。

- `QueryStringMapping`

  可用于将请求的媒体类型映射到查询字符串变量。例如，URL http://localhost/issues?format=atom 中的格式变量会映射到媒体类型 `atom`。

- `UriPathExtensionMapping`

  可用于将 URI 中的路径映射到媒体类型。例如，http:// localhost/issues.atom 中的路径 .atom 会映射到媒体类型 `atom`。

- `RequestHeaderMapping`

  将请求标头映射到媒体类型。如果你不想使用任何标准 HTTP 请求标头，就可以使用这个类。

媒体类型映射通过格式化程序的构造函数注入到格式化程序中。示例 13-19 展示了如何修改格式化程序的构造函数，使用 `QueryStringMapping` 实例，在查询字符串中搜索媒体类型。

示例 13-19：从查询字符串进行媒体类型映射

```
public const string Atom = "application/atom+xml";
public const string Rss = "application/rss+xml";

public SyndicationMediaTypeFormatter()
  : base()
{
  this.SupportedMediaTypes.Add(new MediaTypeHeaderValue(Atom));
  this.SupportedMediaTypes.Add(new MediaTypeHeaderValue(Rss));

  this.MediaTypeMappings
        .Add(new QueryStringMapping("format", "atom",
                new MediaTypeHeaderValue(Atom))); // <1>
}
```

如果格式化程序找到一个查询字符串变量 `format`，变量值为 `atom`，就将其映射到媒体类型 `Atom`（`application/atom+xml`）<1>。

13.3.6 HttpParameterBinding的默认选择

在默认情况下，模型绑定基础结构会尝试使用 FormatterParameterBinder 进行复杂类型参数的绑定，对简单的 .NET 类型则使用 ModelBindingParameterBinder。在有多个复杂类型参数的情况下，除非你使用 FromUriAttribute 属性，明确指定一个参数必须从 URL 中进行绑定，或用 FromBodyAttribute 属性指定参数必须从正文中绑定，否则绑定将会失败。你可以使用 FromBodyAttribute，对指定的参数强制使用 FormatterParameterBinder。如果一个操作包含多个复杂参数，其中只有一个参数可以通过使用 FromBodyAttribute，从请求正文中读取。否则，运行时将会抛出异常。

13.4 模型验证

模型验证是带有模型绑定基础结构的 Web API 提供的另一个功能。你可以使用这个功能强制执行业务规则，或者确保客户端发送数据的正确性。模型验证在模型绑定时在单一位置执行，这种集中性使得代码更易维护和测试。

模型验证的另一个重要功能是，将客户端发送数据的任何可能错误通知客户端，提供更正这些错误的机会。在实践中，如果一个 Web API 没有实现这个功能，开发者会停止在应用程序中使用这个 API。

和模型绑定基础结构的其他部分一样，ASP.NET Web API 中的模型验证也是完全可扩展的。框架自带一个通用的验证程序，使用属性进行模型验证。这个验证程序可以满足大部分情况的需求，并重用了 System.ComponentModel.DataAnnotations 命名空间中的数据标记属性。System.ComponentModel.DataAnnotations 命名空间中提供几种验证属性，例如：Required 标记属性为必需的，RegularExpression 使用正则表达式验证属性值。你也可以自己创建定制的数据标记属性，以满足内建属性没有覆盖到的需求。

13.4.1 将数据标记属性用于模型

假设我们希望对问题模型进行一些验证。我们可以首先使用数据标记属性修饰模型，无需编写很多代码就可以执行通常的验证，更重要的是，使用数据标记属性不需要重复编码。示例 13-20 展示了带有一些数据标记属性的问题模型。

示例 13-20：带有数据标记属性的问题模型

```
public class Issue
{
  [DisplayName("Issue Id")]
  [Required(ErrorMessage = "The issue id is required")]
  [Range(1, 1000, ErrorMessage = "The unit price must be between {1} and {2}")]
  public int Id { get; set; }
```

```
    [DisplayName("Issue Name")]
    [Required(ErrorMessage = "The issue name is required")]
    public string Name { get; set; }

    [DisplayName("Issue Description")]
    [Required(ErrorMessage = "The issue description is required")]
    public string Description { get; set; }
  }
```

这个模型中的所有属性都带有标签，明确声明了各自的用途。一些属性，例如：Name 和 Description，标记为 required；Id 标记为需要指定范围内的值。DisplayName 属性不用于数据验证，但是会影响输出消息的显示。

13.4.2 查询验证结果

一旦模型绑定基础结构基于模型上定义的属性，完成了模型的验证，验证结果就可以由控制器使用，或者以更为集中的方式，由筛选器使用。

要检查模型是否成功绑定，所有验证结果是否通过，最简单的方法是在操作实现中加入几行代码（参见示例 13-21）。

示例 13-21：在操作实现中检查验证结果

```
public class ValidationError
{
  public string Name { get; set; }
  public string Message { get; set; }
}

public class IssueController : ApiController
{
  public HttpResponseMessage Post(Issue product)
  {
    if(!this.ModelState.IsValid)
    {
      var errors = this.ModelState // <1>
               .Where(e => e.Value.Errors.Count > 0)
               .Select(e => new ValidationError // <2>
               {
                    Name = e.Key,
                    Message = e.Value.Errors.First().ErrorMessage
                }).ToArray();

      var response = new HttpResponseMessage(HttpStatusCode.BadRequest);
      response.Content = new ObjectContent<ValidationError[]>(errors,
              new JsonMediaTypeFormatter());

      return response;
    }

    // 操作代码
```

```
    }
}
```

正如示例 13-21 所示，控制器操作的 `ModelState` 属性 <1> 可以得到所有的验证结果。在示例中，操作方法只是简单地将所有的验证错误转换为一个新的模型类 `ValidationError`<2>，我们可以将这个类序列化为 JSON 格式的响应正文，加上一个 `Bad Request` 状态码返回。

这个示例中的代码很通用，你可能希望在多个操作中重用，因此也许最好的方法是把这些通用代码移到一个定制的筛选器中。示例 13-22 中的筛选器包含了同样的检查代码。

示例 13-22：执行验证的 `ActionFilterAttribute` 实现

```
public class ValidationActionFilter : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpActionContext actionContext)
    {
        if (!actionContext.ModelState.IsValid)
        {
          var errors = this.ModelState
                .Where(e => e.Value.Errors.Count > 0)
                .Select(e => new ValidationError
                {
                    Name = e.Key,
                    Message = e.Value.Errors.First().ErrorMessage
                }).ToArray();

          var response = new HttpResponseMessage(HttpStatusCode.BadRequest);
          response.Content = new ObjectContent<ValidationError[]>(errors,
                  new JsonMediaTypeFormatter());

          actionContext.Response = response;
        }
    }
}
```

你可以看到，这个筛选器的实现也非常简单。当筛选器检测到无效模型，执行管道就会自动中断，向 API 使用者发送一个包含验证错误信息的新响应。例如，如果我们发送了一个包含空产品名和无效单价的消息，就会得到示例 13-23 中的响应。

示例 13-23：无效请求消息和对应的响应消息

```
Request Message in JSON

POST http://../Isssues HTTP/1.1
Content-Type: application/json

{
  Id: 1,
  "Name":"",
  "Description": "My issue"
}
```

```
Response Message

HTTP/1.1 400 Bad Request
Content-Type: text/plain; charset=utf-8

[{
  "Message": "The Issue Name is required.",
  "Name": "Name"
}]
```

13.5 小结

模型绑定基础结构是 HTTP 消息和模型对象实例之间的映射层，主要依赖 `HttpParameterBinding` 组件，将参数绑定到 HTTP 消息的各个部分，例如：标头、查询字符串或正文文本。ASP.NET Web API 框架自身提供了 `HttpParameterBinding` 的两个实现：`ModelBindingParameterBinder` 和 `FormatterParameterBinder`，前者使用借用自 ASP.NET MVC 中的传统绑定机制（模型由 HTTP 消息中的片段组成），后者使用格式化程序将媒体类型格式转换成模型。

`ModelBindingParameterBinder` 使用了 `IValueProvider` 实例，从 HTTP 消息的不同部分收集数据，并使用 `IModelBinder` 实例将所有数据组合形成单个模型。

`FormatterParameterBinder` 是内容协商的基础组成部分，因为 `FormatterParameterBinder` 理解如何使用格式化程序，将一个带有语义规则和指定媒体类型的 HTTP 消息正文转换成一个模型。

格式化程序派生自基类 `MediaTypeFormatter`，通常处理单个媒体类型。除了进行参数绑定，模型绑定基础结构还提供一个扩展点，用于在模型反序列化后对其进行验证。默认情况下，模型验证的规则由数据标记属性定义。

第14章 HttpClient

工欲善其事，必先利其器。

本章深入探讨第 10 章中介绍的 System.Net.Http 程序库中的 HttpClient 类。

2009 年初，HttpClient 与 REST Starter Kit（RSK）绑定在一起，首次出现在 CodePlex 上。HttpClient 引入了一些概念，例如：请求 / 响应管道，与请求 / 响应以及强类型标头不同的 HTTP 有效载荷抽象。虽然 .NET 框架 4.0 包含了 RSK 的很多内容，但是没有包含 HttpClient。当 2010 年 Web API 项目启动时，项目的核心任务之一就是重写 HttpClient。

14.1 HttpClient类

简单的事情应该简单实现，HttpClient 坚守这一原则。请看下面的代码：

```
var client = new HttpClient();
string rfc2616Text =
await client.GetStringAsync("http://www.ietf.org/rfc/rfc2616.txt");
```

这个示例初始化了一个新的 HttpClient 对象，发起一个 HTTP GET 请求，并将响应的内容转换成 .NET 字符串。

这段看似寻常的代码为我们提供了足够的背景，讨论与 HttpClient 类的使用相关的一系列问题。

14.1.1 生存周期

虽然 HttpClient 间接实现了 IDisposal 接口，但是我们不推荐在每个请求后释放 HttpClient。这个 HttpClient 对象应该用于你的应用程序所有的 HTTP 请求。如果多个请求使用同一个 HttpClient 对象，那么我们就可以在这个对象中设置 DefaultRequestHeaders，以免像使用 HttpWebRequest 时一样，对每个请求都进行像 CredentialCache 和 CookieContainer 这样的设置。

14.1.2 封装类

有趣的是，HttpClient 自身并不执行发起 HTTP 请求的具体工作，而是将这些工作交给一个派生自 HttpMessageHandler 的聚合对象。HttpClient 的默认构造函数负责初始化聚合对象。你也可以在构造函数中传入一个聚合对象，代码如下所示：

```
var client = new HttpClient((HttpMessageHandler) new HttpClientHandler());
```

HttpClientHandler 在内部使用 System.Net 的 HttpWebRequest 和 HttpWebResponse 类。这种设计提供了最优的结果。今天，我们获得了新接口，访问一个已证明的 HTTP 栈；明天，我们可以将 HttpClientHandler 替换为某些改进过的 HTTP 内部实现，而应用程序的接口不会发生变化。HttpClientHandler 的实现使用了 System.Net 程序库各个版本中通用的部分，可以在多个平台上使用，例如：WinRT 和 Windows Phone。由于这种设计选择，我们无法直接使用一些功能，例如：客户端缓存、管道和客户端认证，因为这些功能都依赖桌面操作系统。要使用这些功能，你需要使用如下代码：

```
var handler = new WebRequestHandler {
                AuthenticationLevel = AuthenticationLevel.MutualAuthRequired,
                CachePolicy = new RequestCachePolicy(RequestCacheLevel.Default)
        };

var httpClient = new HttpClient(handler);
```

WebRequestHandler 类派生自 HttpClientHandler，但位于另一个单独的程序集 System.Net.Http.WebRequest。

HttpClient 实现 IDisposable 接口是为了释放 HttpMessageHandler，HttpMessageHandler 随之尝试关闭底层的 TCP/IP 连接。这意味着，创建一个新的 HttpClient 并发起新的请求，需要创建一个新的底层端口连接，如果只是为了发起一个请求，这种操作的代价可谓非常高昂。

有一点要注意，如果你在 HttpClient 外部实例化一个 HttpMessageHandler，并将这个处理程序传递给 HttpClient 的构造程序，那么释放 HttpClient 就会导致这个处理程序无法使用。如果配置这个处理程序相当复杂，那么你可能会希望能够在多个 HttpClient 实例中重

用这个处理程序。幸好，ASP.NET Web API 添加了一个新的 HttpClient 构造函数，以支持处理程序的重用：

```
var handler = HttpHandlerFactory.CreateExpensiveHandler(};
var httpClient = new HttpClient(handler, disposeHandler:false);
```

如果你在构造函数中告诉 HttpClient 不要释放 HttpMessageHandler，就可以在多个 HttpClient 实例中重用这个消息处理程序。

14.1.3　多个实例

一旦发出第一个请求，HttpClient 的某些属性就无法再修改，这可能导致你想要创建多个 HttpClient 实例。这些属性有：

```
public Uri BaseAddress
public TimeSpan Timeout
public long MaxResponseContentBufferSize
```

14.1.4　线程安全

HttpClient 类是线程安全的，可以很愉快地管理多个并行的 HTTP 请求。如果前面列出的三个属性在请求处理过程中发生了变化，那么可能导致难以定位的错误。

这些属性比较容易理解，但是我们需要强调一下 MaxResponseContentBufferSize。MaxResponseContentBufferSize 属性的数据类型是 long，但是默认值和最大值仅为 Int32.MaxValue，这个最大值对大多数情况都是足够的。不要担心，尽管这个属性值最大可以设置到 4 GB，但是 HttpClient 只会分配 HTTP 有效载荷缓存所需的内存，不会过多分配。

HttpClient 不仅是 HttpMessageHandler 的封装类，配置属性的宿主，能够保存日志消息，而且还提供一些辅助方法，使发送常用请求的工作更加容易。有了这些辅助方法，HttpClient 可以完全取代 System.Net.WebClient。

14.1.5　辅助方法

本章的第一个示例代码使用了 GetStringAsync 方法，除此之外还有 GetStreamAsync 和 GetByteArrayAsync 方法。所有这些辅助方法名都以 Async 结尾，说明这些是异步方法，这些方法的返回值都是 Task 对象。因此，我们可以在支持 async 和 await 的平台上使用这些关键字。.NET 4.5 的原则是将所有执行时间超过 50 毫秒的方法定义为异步方法，这是为了鼓励开发者在设计应用程序时，不要阻塞用户界面线程，以创建响应性更好的应用。我们的示例使用了返回任务的 .Result 属性，以阻塞调用线程，返回字符串结果。这种方法规避了 .NET 4.5 推荐的原则，会带来一些问题，我们将在本章稍后进行讨论。但是，为了简单起见，我们将使用 .Result 来模拟同步请求。

14.1.6 抽丝剥茧

HttpClient 的辅助方法虽然使用方便，但是我们无法窥见其内部的操作。如果我们去除一层简化，就得到了 GetAsync 方法，使用方法如下：

```
var client = new HttpClient();
HttpResponseMessage response;
response = await client.GetAsync("http://www.ietf.org/rfc/rfc2616.txt");
HttpContent content = response.Content;
string rfc2616Text = await content.ReadAsStringAsync();
```

在这段示例代码中，我们可以访问响应对象和内容对象。使用这些对象，我们可以检查 HTTP 标头中的元数据，以决定如何使用返回的内容。

14.1.7 完成的请求无异常

使用 HttpClient 时，在默认情况下，已完成的 HTTP 请求不会抛出异常，这与 HttpWebRequest 的行为不同。使用 HttpClient 时，如果传输层出现错误，HttpClient 可能会抛出异常，但是 3xx、4xx 和 5xx 这样的状态码并不会导致异常。而使用 HttpWebRequest 时，3xx、4xx 和 5xx 这样的状态码也会导致异常。我们可以使用 HttpResponseMessage 的 IsSuccessStatusCode 属性来判断状态码是否为 2xx，如果状态码为不成功，就可以使用 EnsureSuccessStatusCode 方法，手工触发一个异常。

HTTP 请求返回的状态码经常可以由应用程序代码直接处理，不一定需要抛出异常。例如，对于很多的 3xx 状态码，我们可以向返回的 location 标头中的 URI 发出第二个请求，自动进行处理；对于 503 Service Unavailable，我们可以使用重试机制，确保暂时的中断不会导致应用程序发生严重错误。本章稍后部分，我们将进一步讨论如何构建客户端，对 HTTP 状态码做出智能响应。

14.1.8 内容为王

HttpContent 是一个抽象类，ASP.NET Web API 框架对其只提供少数具体实现。HttpContent 对如何处理发送字节的细节进行了抽象。HttpContent 实例负责处理如何刷新和定位流，分配和释放内存，以及将 CLR 类型转换成用于传输的字节，还可以用于访问与 HTTP 有效载荷相关的标头。

你可以使用第 10 章介绍过的 ReadAs 方法，访问一个 HttpContent 对象的内容。虽然 HttpContent 的抽象机制使你无需关心读取网络传输字节的细节，但是有一个容易忽视的关键细节，需要注意。HttpClient 的一些方法带有一个 completionOption 参数。这个 completionOption 参数决定了异步任务在收到响应标头后就完成，还是在将完整的响应正文读入缓冲区后才完成。

你可能会希望在获得标头之后就将让任务完成，原因如下：

- 客户端可能无法理解响应的媒体类型，如果使用的是计费网络，下载正文会浪费时间和金钱；
- 你可能希望在下载响应内容的同时，根据响应标头数据进行一些处理。

下面的代码是一个假设的示例，演示如何使用这一参数。

```
var httpClient = new HttpClient();
httpClient.BaseAddress = new Uri("http://www.ietf.org/rfc/");
var tcs = new CancellationTokenSource();

var response = await httpClient.GetAsync("rfc2616.txt",
        HttpCompletionOption.ResponseHeadersRead, tcs.Token);
// 标头已经返回

if (!IsSupported(response.Content.ContentType)) {
        tcs.Cancel();
        return;
}
UIManager userInterfaceManager = new UIManager();

// 开始根据内容类型构建适当的 UI
userInterfaceManager.PrepareTheUI(content.ContentType);

// 开始从网络拉取有效载荷数据
var payload = await response.Content.ReadAsStreamAsync()

// 有效载荷获取完毕
userInterfaceManager.Display(payload);
```

要实现这一技术，`UIManager` 必须进行一些线程同步控制，因为 `Display` 方法和 `PrepareTheUI` 方法很可能由不同的线程调用，`Display` 方法可能需要等到 UI 准备好才能开始执行。有时，为了能够有效地同时执行两项任务，以提高性能，这种额外的设计工作是值得的。当然，如果你的客户端必须解析有效载荷才能决定要显示什么内容，那么这种技术就没有什么用处了。

14.1.9　取消请求

我们要讨论的 `GetAsync` 方法的最后一个参数是 `CancellationToken`。通过创建一个 `CancellationToken` 并传给 `GetAsync` 方法，发起调用的对象可以有机会取消这个 `Async` 操作。请注意，取消操作会导致 `Async` 抛出异常，你要准备好捕获这个异常。

在下面的示例中，如果一个请求不能在一秒钟内完成，代码就会取消这个请求。这段代码只演示了如何使用 `Cancel` 方法，因为 `HttpClient` 自身提供超时机制。

```
[Fact]
```

```
public async Task RequestCancelledByCaller()
{
    Exception expectedException = null;
    bool done = false;

    var httpClient = new HttpClient();
    var cts = new CancellationTokenSource();

    var backgroundRequest = new TaskFactory().StartNew(async () =>
    {
        try
        {
            var request = new HttpRequestMessage()
            {
                RequestUri = new Uri("http://example.org/largeResource")
            };

            var response = await httpClient.SendAsync(request,
                HttpCompletionOption.ResponseHeadersRead, cts.Token);

            done = true;
        }
        catch (TaskCanceledException ex)
        {
            expectedException = ex;
        }
    }, cts.Token);

    // 等待请求完成
    Thread.Sleep(1000);

    if (!done)
        cts.Cancel();

    Assert.NotNull(expectedException);
}
```

14.1.10 SendAsync

到目前为止我们介绍的 HttpClient 的所有方法都是对单个方法 SendAsync 的简单封装，SendAsync 的方法签名如下：

```
public Task<HttpResponseMessage> SendAsync(
      HttpRequestMessage request,
      HttpCompletionOption completionOption,
      CancellationToken cancellationToken)
```

通过创建一个 HttpRequestMessage，并设置其 Method 属性和 Content 属性，你可以轻易使用 SendAsync 复制 HttpClient 辅助方法的行为。但是，HttpRequestMessage 只能使用一次。在发送请求后，HttpRequestMessage 就会立即释放，以确保任何相关联的 Content 对象也得到释放。在很多情况下，这种做法应该并非必要，但是，如果一个 HttpContent 对象封装了一个只能向前访问的流，那么不重新初始化这个流就无法重新发送这个内容，而

HttpContent 对象并没有提供这样的接口。要规避这个限制，我们可以引入一个链接类作为请求工厂（详见第 9 章）。

```
var httpClient = new HttpClient();
httpClient.BaseAddress = new Uri("http://www.ietf.org/rfc/");

var request = new HttpRequestMessage() {
        RequestUri = new Uri("rfc2616.txt"),
        Method = HttpMethod.Get
}
var response = await httpClient.SendAsync(request,
        HttpCompletionOption.ResponseContentRead, new CancellationToken());
```

SendAsync 方法是 HttpClient 和 Web API 架构的核心部分。SendAsync 是 HttpMessage Handler 使用的主要方法，而 HttpMessageHandler 是请求和响应管道的组件。

14.2　客户端消息处理程序

消息处理程序（message handler）是 HttpClient 和 Web API 的核心架构组件之一。无论是在客户端还是在服务器上，每个请求和响应消息都会经过一个类“链”，链中的每个类都派生自 HttpMessageHandler。对于 HttpClient，默认情况下这个链中只有一个处理程序：HttpClientHandler。你可以在这个链的开头插入更多的 HttpMessageHandler 实例，对默认行为进行扩展，示例 14-1 对此进行了演示。

示例 14-1：向客户端请求管道添加处理程序

```
var customHandler = new MyCustomHandler()
      { InnerHandler = new HttpClientHandler()};

var client = new HttpClient(customHandler);

client.GetAsync("http://example.org",content);
```

示例 14-1 中的代码创建的对象如图 14-1 所示。

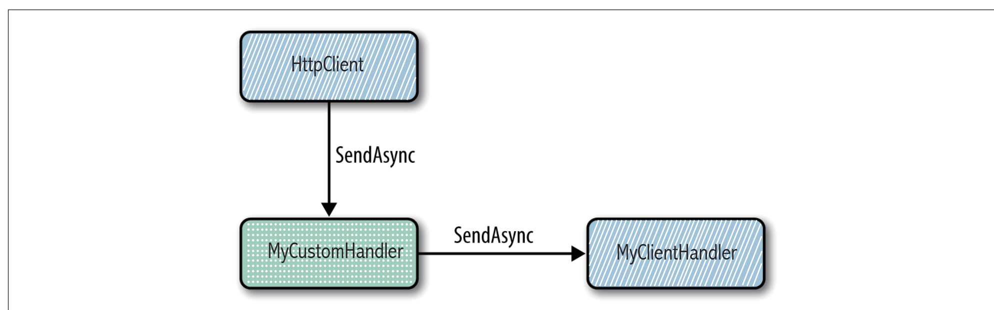


图 14-1：HttpMessageHandler 的扩展性

多个消息处理程序可以链接在一起，提供更多的功能。但是，基类 HttpMessageHandler 自身没有提供链接能力，其派生类 DelegatingHandler，提供了 InnerHandler 属性，以支持链接功能。

示例 14-2 展示了当服务器不支持 PUT 和 DELETE 方法，并且要求请求消息包含 X-HTTP-Method-Override 标头时，如何使用消息处理程序，使客户端能够对服务器执行 PUT 和 DELETE 操作。

示例 14-2：HttpMethodOverrideHandler

```
public class HttpMethodOverrideHandler: DelegatingHandler
    {
        protected override Task<HttpResponseMessage> SendAsync(
                HttpRequestMessage request,
                System.Threading.CancellationToken cancellationToken)
        {
            if (request.Method == HttpMethod.Put) {
                request.Method = HttpMethod.Post;
                request.Headers.Add("X-HTTP-Method-Override", "PUT");
            }
            if (request.Method == HttpMethod.Delete)
            {
                request.Method = HttpMethod.Post;
                request.Headers.Add("X-HTTP-Method-Override", "DELETE");
            }
            return base.SendAsync(request, cancellationToken);
        }
    }
```

DelegatingHandler 的派生类 MessageProcessingHandler（参见示例 14-3），进一步简化了处理程序的创建，但只限于定制行为不需要执行长时间异步操作的情况。

示例 14-3：MessageProcessingHandler

```
public class HttpMethodOverrideMessageProcessor : MessageProcessingHandler {

     protected override HttpRequestMessage ProcessRequest(
             HttpRequestMessage request,
             CancellationToken cancellationToken) {

         if (request.Method == HttpMethod.Put)
         {
             request.Method = HttpMethod.Post;
             request.Headers.Add("X-HTTP-Method-Override", "PUT");
         }
         if (request.Method == HttpMethod.Delete)
         {
             request.Method = HttpMethod.Post;
             request.Headers.Add("X-HTTP-Method-Override", "DELETE");
         }
         return request;
     }

     protected override HttpResponseMessage ProcessResponse(
```

```
            HttpResponseMessage response,
            CancellationToken cancellationToken) {
        return response;
    }
}
```

使用这些消息处理程序进行功能扩展时，你需要注意，这些处理程序的执行线程经常会与发起请求的线程不同。如果这个处理程序尝试切换到请求线程——例如：回到 UI 线程去更新某个用户界面的空间——那么可能产生死锁。如果最初的请求是阻塞的，等待响应返回，那么消息处理程序切回请求线程就会导致死锁。你可以使用 .NET 4.5 的 `async await` 机制避免这种问题，但尽量不要使用 `.Result` 模拟同步请求。

14.2.1　代理处理程序

`HttpMessageHandler` 有很多潜在的用途，其中之一是用做操控发出请求的代理。下面的示例是 Runscope 调试服务的一个代理。

```
public class RunscopeMessageHandler : DelegatingHandler
{
    private readonly string _bucketKey;

    public RunscopeMessageHandler(string bucketKey,
            HttpMessageHandler innerHandler)
    {
        _bucketKey = bucketKey;
        InnerHandler = innerHandler;
    }

    protected override Task<HttpResponseMessage> SendAsync(
            HttpRequestMessage request,
            CancellationToken cancellationToken)
    {
        var requestUri = request.RequestUri;
        var port = requestUri.Port;

        request.RequestUri = ProxifyUri(requestUri, _bucketKey);
        if ((requestUri.Scheme == "http" && port != 80 )
                ||  requestUri.Scheme == "https" && port != 443)
        {
            request.Headers.TryAddWithoutValidation(
                "Runscope-Request-Port", port.ToString());
        }
        return base.SendAsync(request, cancellationToken);
    }

             private Uri ProxifyUri(Uri requestUri,
                             string bucketKey,
                             string gatewayHost = "runscope.net")
    {
     ...
    }
}
```

在这个示例中，消息处理程序将请求 URI 修改为指向代理，而非原始资源。

14.2.2 伪响应处理程序

你可以使用消息处理程序，辅助客户端代码的测试。如果创建如下的消息处理程序：

```
public class FakeResponseHandler : DelegatingHandler
    {
        private readonly Dictionary<Uri, HttpResponseMessage> _FakeResponses
            = new Dictionary<Uri, HttpResponseMessage>();

        public void AddFakeResponse(Uri uri, HttpResponseMessage responseMessage)
        {
            _FakeResponses.Add(uri,responseMessage);
        }

        protected async override Task<HttpResponseMessage> SendAsync(
                HttpRequestMessage request,
                CancellationToken cancellationToken)
        {
            if (_FakeResponses.ContainsKey(request.RequestUri))
            {
                return _FakeResponses[request.RequestUri];
            }
            else
            {
                return new HttpResponseMessage(HttpStatusCode.NotFound)
                    { RequestMessage = request}
            }
        }
    }
```

你可以用这个消息处理程序替换 HttpClientHandler：

```
[Fact]
public async Task CallFakeRequest()
{
    var fakeResponseHandler = new FakeResponseHandler();
    fakeResponseHandler.AddFakeResponse(
        new Uri("http://example.org/test"),
        new HttpResponseMessage(HttpStatusCode.OK));

    var httpClient = new HttpClient(fakeResponseHandler);

    var response1 = await httpClient.GetAsync("http://example.org/notthere");
    var response2 = await httpClient.GetAsync("http://example.org/test");

    Assert.Equal(response1.StatusCode,HttpStatusCode.NotFound);
    Assert.Equal(response2.StatusCode, HttpStatusCode.OK);
}
```

为了测试客户端服务，你必须确保客户端允许进行 HttpClient 实例的注入。这也是为什么我们推荐共享一个 HttpClient 实例，而不是对每个请求临时实例化 HttpClient。FakeResponseHandler 需要预先填充预期接收的响应。使用这种设置，我们可以模拟服务器链接，进行客户端代码测试：

```
[Fact]
public async Task ServiceUnderTest()
{
    var fakeResponseHandler = new FakeResponseHandler();
    fakeResponseHandler.AddFakeResponse(
            new Uri("http://example.org/test"),
            new HttpResponseMessage(HttpStatusCode.OK)
                    {Content = new StringContent("99")});
    var httpClient = new HttpClient(fakeResponseHandler);

    var service = new ServiceUnderTest(httpClient);
    var value = await service.GetTestValue();

    Assert.Equal(value, 99);
}
```

14.2.3　创建可以重用的响应处理程序

在第 9 章中，我们讨论了*响应式客户端*（reactive client）的概念，这种客户端解除了响应处理与请求上下文之间的耦合。

消息处理程序和 `HttpClient` 管道是实现响应式客户端的自然解决方法，还可以顺便简化发出请求的过程。

请看示例 14-4 中的消息处理程序。

示例 14-4：可插入的响应处理程序

```
public abstract class ResponseAction
    {
        abstract public bool ShouldRespond(
                ClientState state,
                HttpResponseMessage response);

        abstract public HttpResponseMessage HandleResponse(
                    ClientState state,
                    HttpResponseMessage response);
    }

public class ResponseHandler : DelegatingHandler
    {
        private static readonly List<ResponseAction> _responseActions
                = new List<ResponseAction>();

        public void AddResponseAction(ResponseAction action)
        {
            _responseActions.Add(action);
        }

        protected override Task<HttpResponseMessage> SendAsync(
                HttpRequestMessage request,
                CancellationToken cancellationToken)
```

```
        {
            return base.SendAsync(request, cancellationToken)
                .ContinueWith<HttpResponseMessage>(t =>
                     ApplyResponseHandler(t.Result));
        }

        private HttpResponseMessage ApplyResponseHandler(
                    HttpResponseMessage response)
        {

            foreach (var responseAction in _responseActions)
            {
                if (responseAction.ShouldRespond(response))
                {
                    var response = responseAction.HandleResponse(response);
                    if (response == null) break;
                }
            }
            return response;
        }
    }
```

在这个示例中，我们创建了一个委托处理程序，如果一个 ResponseAction 的 ShouldResponse 返回 ture，就将响应分发到这个 ResponseAction 类。使用这种机制，我们可以定义和插入任意多个响应操作。ShouldResponse 方法可以简单地查看 HTTP 状态码，或者实现复杂的逻辑，查看内容类型，甚至解析有效载荷以寻找特定的标记。

发出 HTTP 请求的过程也得到了简化，如示例 14-5 所示。

示例 14-5：使用响应处理程序

```
var responseHandler = new ResponseHandler()
        {InnerHandler = new HttpClientHandler()};

responseHandler.AddAction(new NotFoundHandler());
responseHandler.AddAction(new BadRequestHandler());
responseHandler.AddAction(new ServiceUnavailableRetryHandler());
responseHandler.AddAction(new ContactRenderingHandler());

var httpClient = new HttpClient(responseHandler);

httpClient.GetAsync("http://example.org/contacts");
```

14.3　小结

HttpClient 是在 .NET 平台上使用 HTTP 的一大进步。通过 HttpClient，我们得到了一个和 WebClient 一样易于使用的接口，而且比 HttpWebRequest/HttpWebResponse 功能更强，更易配置。这个接口还支持未来的协议实现。使用 HttpClient，测试变得更加容易，其管道架构使我们可以实现很多横切关注点，还依然保持易用性。

第 15 章 安全

穿袈裟的不一定是和尚。

从广义上说，计算机系统安全包括了很多主题和技术，从加密算法一直到系统可用性和灾难恢复系统，都属于计算机安全的范畴。本章并不会讨论这么多的内容。我们将关注与 Web API 密切相关的安全问题——具体而言，就是传输安全、身份验证和授权。因此，在接下来的小节中，我们将从理论和实践的角度，以 ASP.NET Web API 为支持技术，解决这些安全问题。

下一章是对本章内容的补充，将只关注 OAuth 2.0 框架，解决基于 HTTP 的 API 中访问控制的一组协议和模式。

15.1 传输安全

传输信息的保密性和完整性是重要的安全需求，在设计和实现分布式系统时必须满足这些需求。糟糕的是，HTTP 协议几乎不提供安全方面的支持。因此，开发者们通常采取的解决办法是，在一个安全传输层上使用 HTTP 协议，如 RFC 1818 定义的“HTTP Over TLS”，形成了我们所知的 HTTPS。简单地说，RFC 1818 规范声明，如果客户端使用 `https` 方案，对一个 URI 执行 HTTP 请求（例如：https://www.example.net），那么这个 HTTP 协议就位于一个安全传输层（TLS 或 SSL）之上，而不是直接在 TCP 层上，如图 15-1 所示。使用这种方式，请求和响应消息在两个传输端点之间进行传送时，都受到传输协议的保护。

RFC 5246 定义的 TLS（Transport Layer Security protocol，传输层安全协议），是 SSL（Secure

Socket Layer protocol，安全套接层协议）的改进版。[1] 这两个协议都是为了在两个通信实体——通常称为*对等方*（peer）——之间，提供一个*安全的*双向连接，这个连接具有如下特征。

- *完整性*（integrity）
  每个对等方都确信接收到的字节流与远端的对等方传送的字节流相同。如果字节流遭到第三方的修改，或者转发，系统都能检测发现，并终止连接。

- *保密性*（confidentiality）
  每个对等方都得到保证，只有远端的对等方能查看发送的字节流。

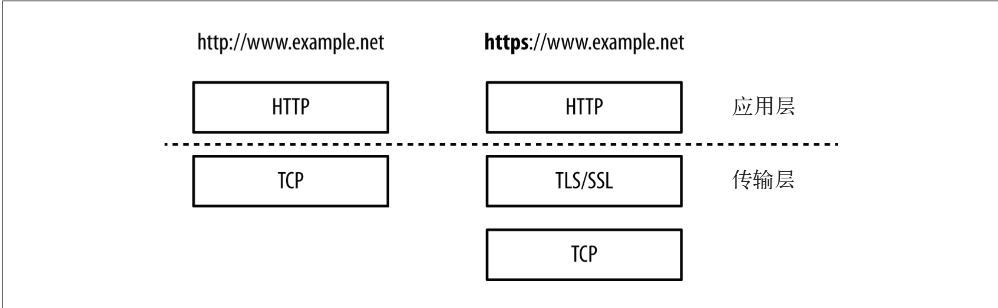


图 15-1：Https 方案和传输安全

除了完整性和保密性，TLS 协议还能够执行对等身份验证，为客户端或服务器提供远端对等方的验证身份。非常重要的一点是，在用于 HTTP 时，TLS 还负责执行服务器身份验证的基本任务，在客户端发送任何请求消息之前，为客户端提供服务器的验证身份。我们将在 15.3 节详细讨论基于 TLS 的身份验证。

TLS 协议自身分为两个主要的子协议。*记录子协议*（record subprotocol）使用对称加密方案和 MAC（Message Authentication Codes，消息认证码），保护字节流的交换，提供完整性和保密性。记录子协议位于一个可靠传输层（如 TCP）之上，由三层组成。第一层将进入的字节流划分为记录，每个记录最长为 16 KB。第二层对每个记录进行压缩。最后一层使用 MAC-then-Encrypt 的方式进行加密保护，即：首先对已加密的记录加上一个序列号，计算出一个 MAC 值，然后对已加密记录和 MAC 值一起进行加密。

握手子协议（handshake subprotocol）用于建立 TLS 操作参数，即记录子协议使用的安全参数（例如：加密算法和 MAC 密钥）。握手子协议支持多种密钥生成技术。但是，在用于 Web 时，最常见的做法是使用基于公用密钥的加密方法和证书。附录 G 对这一内容进行了简要的介绍，并展示了如何创建在开发环境中使用的密钥和证书。

注 1：在本章余下部分，我们将二者都称为 TLS。

15.2　在ASP.NET Web API中使用TLS

TLS 协议运行于传输层之上，这意味着 TLS 协议是由低层的 HTTP 托管基础结构实现的，在 Windows 上就是内核模式的 HTTP.SYS 驱动程序。因此，TLS 相关的大多数配置是在 ASP.NET Web API 之外完成的，而且 IIS 托管方式和自托管方式下的 TLS 配置也有所不同。

15.2.1　IIS托管时使用TLS

在 IIS 上，TLS 是通过给站点添加 HTTPS 绑定进行配置的，如图 15-2 所示。

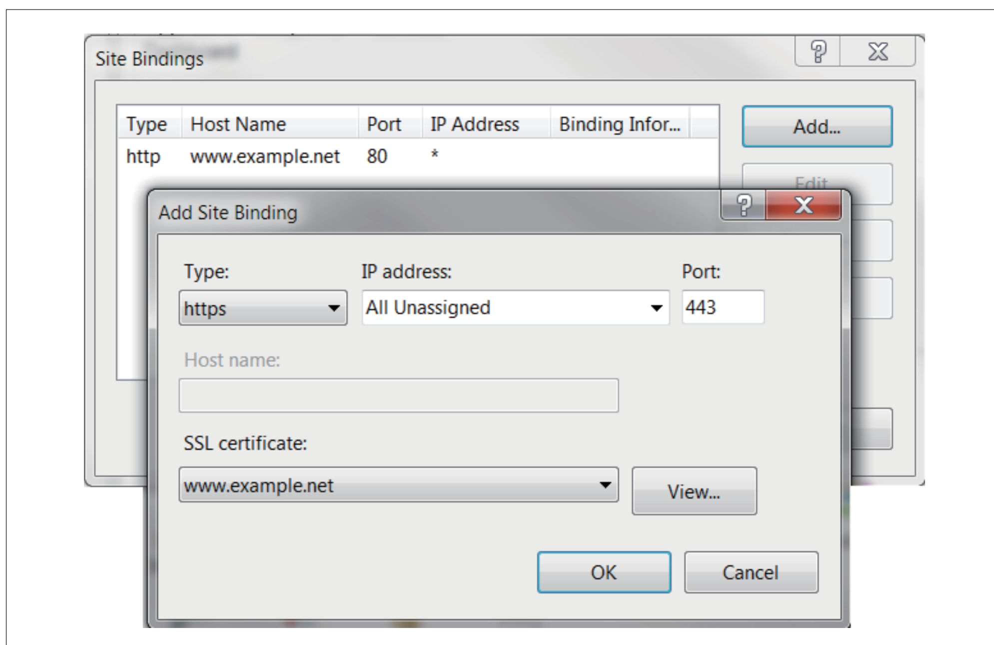


图 15-2：给一个站点添加 HTTPS 绑定

添加 HTTPS 绑定由服务器证书进行配置，这个证书必须安装在本地计算机的个人存储，有一个相关的私有密钥，并有一个有效证书路径，指向一个信任的根证书颁发机构。除此之外，你不需要对 IIS 配置或 Web API 配置进行任何修改。

图 15-3 展示了浏览器通过 HTTPS 执行请求的用户界面。请注意，界面上的信息既有服务器的标识（www.example.net），也有证书颁发机构的名称（“Demo Certification Authority”）。

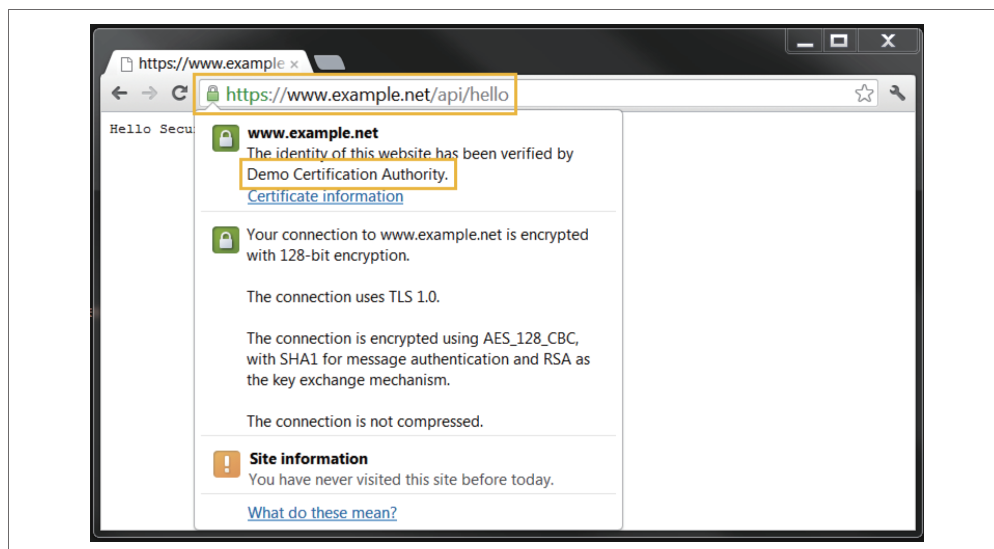


图 15-3：使用 HTTPS 访问 ASP.NET Web API

在 IIS 7.5 中，多个站点的 HTTP 绑定可以配置为指向同一个 IP 地址和端口，因为请求解复用（demultiplexing，对选中的站点进行请求分发）会使用请求的 `Host` 标头中的主机名。但是，使用不同证书的多个 HTTPS 绑定不能配置为执行同一个 IP 地址和端口（因为早在收到 HTTP 请求之前，建立 TLS 连接时就需要使用证书）。因此，要在一个服务器上托管多个 HTTPS 站点，我们可以采取如下方法。

- 为每个 HTTPS 绑定使用不同的 IP 地址或端口。
- 对所有绑定使用同一个证书，这通常意味着在这个证书的主体名中使用通配符。你也可以选择使用主体别名（Subject Alternative Name）扩展，为同一个证书定义多个主体名。

主体别名

X.509 证书规范定义了一个扩展字段主体别名（Subject Alternative Name），允许在一个证书中包含一个或多个主体名。例如，在这本书编写之时，到 https://www.google.com 的连接使用一个包含 44 个别名的 X.509 服务器证书，其中有：*.google.com、*.android.com、*.google.com.ar、*.google.ca，以及 *.google.pt。

RFC 4366 定义了一个新的 TLS 扩展，名为 SNI（Server Name Indication，服务器名称标识），这一扩展将 HTTP 主机名添加到 TLS 的初始握手通信中。使用这一额外的信息，即便建立的 TCP 连接指向同一个 IP 地址和端口，服务器也可以为每个主机名使用不同的证书。遗憾的是，只有 IIS 8.0 或更高版本才支持 SNI，IIS 7.5 或较低版本都不提供这一功能。

15.2.2　自托管时使用TLS

当使用自托管时，你可以使用命令行工具 `netsh`，进行 TSL 配置：

```
netsh http add sslcert ipport=0.0.0.0:port certhash=thumbprint appid={app-guid}
```

其中：

- `ipport` 是服务监听的 IP 地址和端口（特殊地址 `0.0.0.0` 匹配本机的任何 IP 地址）；
- `certhash` 是服务器证书的 SHA-1 散列值的十六进制表示；
- `appid` 只是用于标识应用程序的一个 GUID。

自托管时所选的服务器证书与 IIS 托管时的要求相同，即：证书必须安装在*本地计算机*的个人存储，有一个相关的私有密钥，并有一个有效证书路径，指向一个信任的根证书颁发机构。唯一的区别是，ASP.NET Web API 配置需要在自托管监听地址中使用 `https` 格式：

```
var config = new HttpSelfHostConfiguration("https://www.example.net:8443");
```

传输安全就介绍到这里了。下一节将介绍身份验证。

15.3　身份验证

根据 RFC 4949（因特网安全词典）的定义，*身份验证*（authentication）是“验证一个系统实体或系统资源具有某种属性值的过程”。对 HTTP 而言，客户端和服务器是两个显而易见的*系统实体*，这两个实体通常都需要进行属性验证。

一方面，我们需要进行*服务器身份验证*，以预先向客户端保证，请求消息只会发送给*正确的*源服务器——即：指定资源所在的服务器或应该创建指定资源的服务器。在这种情况下，消息的发送者需要在发送消息之前，对消息的接收者进行身份验证，通常是认证传输连接的另一端。服务器身份验证也需要检验收到的响应消息，是否确实由*正确的*服务器产生。与客户端交互的资源由 URI 标识，因此身份验证过程要检验的主要属性就是 URI 中主机名（IP 地址或 DNS 名）的所有权。

另一方面，*客户端身份验证*为服务器提供身份信息，用于判断请求消息是否应该得到授权——即：所请求的方法是否适用于指定的资源。在这种情况下，身份验证过程要检验的属性与上下文有关，可能是简单的不透明标识符，如用户名，也可能是一个属性集合，例如：电子邮件、姓名、角色、地址，以及银行账号和社会安全号码。

我们在随后的几节中将会看到，这些身份验证需求可以在两个层次上实现：

- 通过在安全连接上发送和接收 HTTP 消息，在传输层实现身份验证；
- 通过在消息上附加用于对身份验证消息源的安全信息，在消息层实现身份验证。

但是，在介绍这些身份验证机制的细节之前，我们要首先了解 .NET 框架如何表示身份——即身份验证过程的输出。

15.3.1 声明模型

.NET 框架从版本 1.0 开始，提供两个表示身份的接口：`IPrincipal` 和 `IIdentity`（参见图 15-4）。`IPrincipal` 接口“表示运行代码的用户的安全上下文”。例如，在 HTTP 请求处理上下文中，`IPrincipal` 接口表示请求消息的产生者——HTTP 客户端。`IPrincipal` 接口提供一个 `IsInRole` 方法，用于查询请求者是否属于指定的角色，还提供一个类型为 `IIdentity` 的 `Identity` 属性。`IIdentity` 接口通过三个属性表示一个用户标识：

- `AuthenticationType` 字符串；
- `Name` 字符串；
- `IsAuthenticated` 字符串。

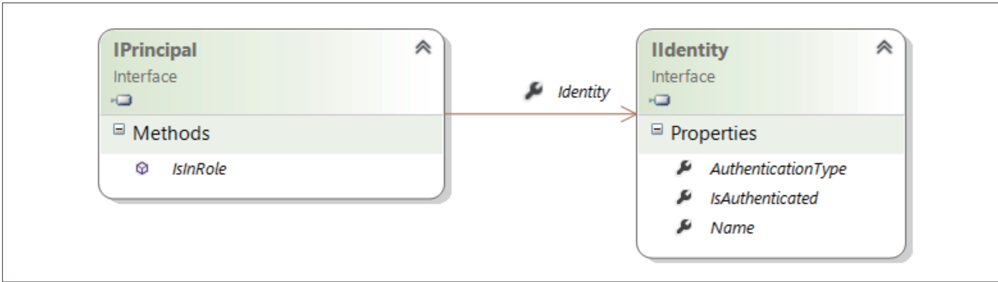


图 15-4：`IPrincipal` 和 `IIdentity` 接口

我们可以通过 `Thread.CurrentPricipal` 静态属性，访问当前用户对象（即运行当前代码的用户的安全上下文对象），也可以通过上下文相关的属性（例如：ASP.NET MVC 的 `System.Web.Mvc.Controller.User` 属 性， 或 WCF 的 `System.ServiceModel.ServiceSecurityContext.PrimaryIdentity`）获得这一信息。在 ASP.NET Web API 上下文中，`ApiController.User` 属性的类型也是 `IPrincipal`，可以用于访问当前主体。

.NET 框架也提供 `IPrincipal` 和 `IIdentity` 接口的一组具体实现：

- `GenericPrincipal`、`WindowsPrincipal` 和 `RolePrincipal` 类实现了 `IPrincipal` 接口；
- `GenericIdentity`、`WindowsIdentity` 和 `FormsIdentity` 类实现了 `IIdentity` 接口。

然而，旧有模型的用户标识概念颇为狭隘，将其限制为一个简单的字符串和一个角色查询方法。而且，这种模型假设存在一个隐含的标识机构，而实际上身份验证信息可以来自多个提供方，既可以是社会站点，也可以是组织目录。

声明模型的设计目标是，基于*声明*（claim）概念，定义表示用户标识的一种新方式，以克

服旧模型的限制。*A Guide to Claims-Based Identity and Access Control*（Microsoft Patterns & Practices）一书将 claim 定义为“一个主体对于自己或其他主体所做的声明，例如：一个名字、身份、密钥、群组、权限或能力”。我们要强调这一定义的两个特征。首先，这个定义非常宽泛，包括了不同的身份属性，从简单的名字标识符到身份验证能力。其次，这个定义明确指出，声明可以由多方做出，被标识的主体也可以做出声明（自声明）。

.NET 框架 4.5 使用这种声明模型表示用于标识，还引入了 `System.Security.Claims` 命名空间，其中包含与声明模型相关的几个类。`Claims` 类（参见图 15-5）由三个核心属性组成：

- `Issuer` 是一个字符串，标识做出标识声明的机构；
- `Type` 是一个字符串，描述声明类型；
- `Value` 也是一个字符串，包含声明值。

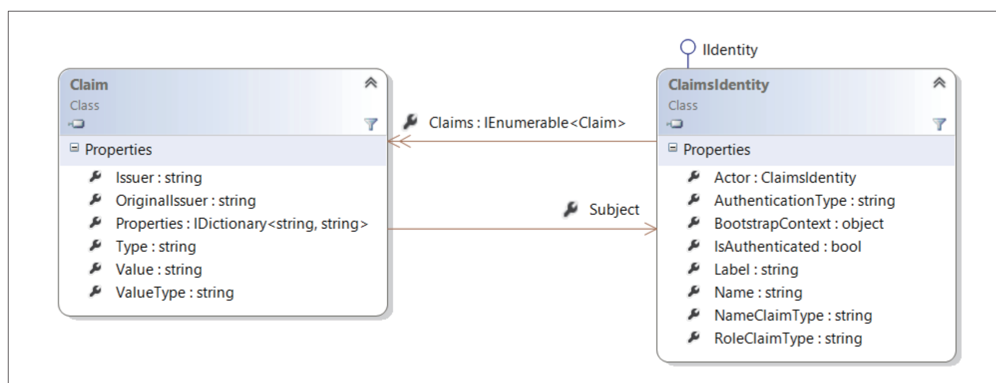


图 15-5：`Claim` 和 `ClaimsIdentity` 类

下面的代码段演示了从进程的 Windows 标识得到的声明的三个 `Claim` 核心属性：

```
[Fact]
public void Claims_have_an_issuer_a_type_and_a_value()
{
    AppDomain.CurrentDomain.SetPrincipalPolicy(
        PrincipalPolicy.WindowsPrincipal);
    var identity = Thread.CurrentPrincipal.Identity as ClaimsIdentity;
    Assert.NotNull(identity);
    var nameClaim = identity.Claims
        .First(c => c.Type == ClaimsIdentity.DefaultNameClaimType);
    Assert.Equal(identity, nameClaim.Subject);

    Assert.Equal("AD AUTHORITY", nameClaim.Issuer);
    Assert.Equal(ClaimTypes.Name, nameClaim.Type);
    Assert.Equal(
        "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name",
        nameClaim.Type);
    Assert.True(nameClaim.Value.EndsWith("pedro"));
}
```

ClaimType 类包含一组常用的声明类型标识符：

```
public static class ClaimTypes
{
    public const string Role = "http://schemas.microsoft.com/.../claims/role";
    public const string AuthenticationInstant = ...
    public const string AuthenticationMethod = ...
    public const string AuthorizationDecision = ...
    public const string Dns = ...
    public const string Email = ...
    public const string MobilePhone = ...
    public const string Name = ...
    public const string NameIdentifier = ...
    // 省略其他成员
}
```

.NET 框架 4.5 还引入了两个新的基于声明的具体类：

- ClaimsIdentity 类将用户标识表示为一个声明序列；
- ClaimsPrincipal 类将用户身份表示为一个或多个基于声明的标识。

请注意，这些新类也实现了原有的 IPrincipal 和 IIdentity 接口，因此可以用于遗留代码。此外，原有的表示用户身份和标识的具体类，例如：WindowsPrincipal 或 FormsIdentity，也经过重新改造，继承这些新的基于声明的类，如图 15-6 所示。

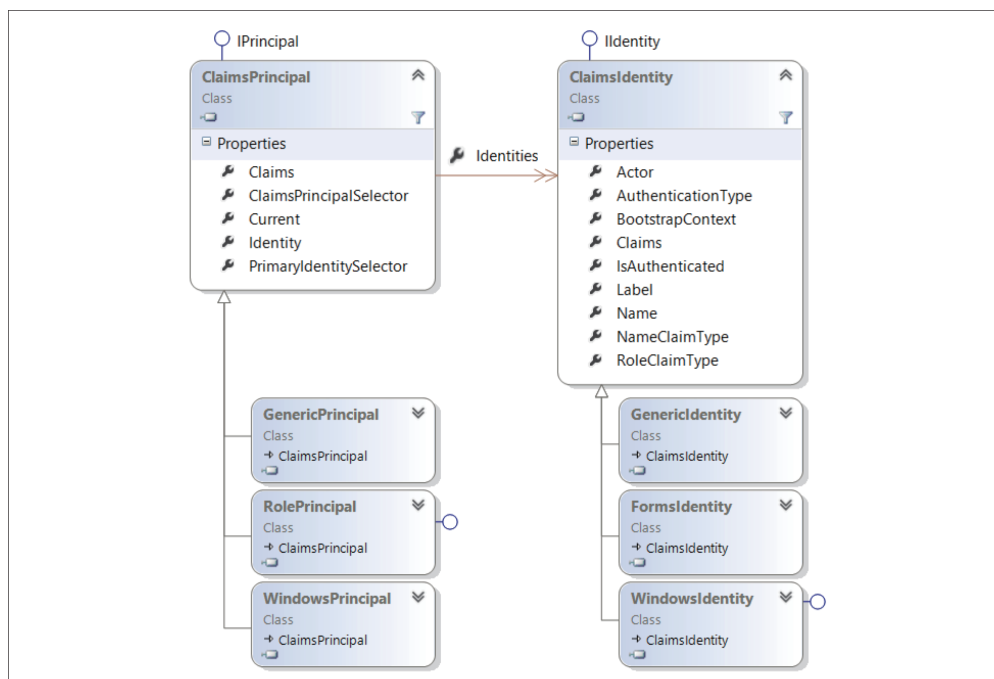


图 15-6：ClaimsPrincipal 类和 ClaimsIdentity 类

在本章余下的部分，我们将用这些新的基于声明的类表示用户标识。

> **Windows 身份验证基础和 .NET 框架**
>
> 具有声明意识的应用程序的类模型最早出现在 2009 年，是 .NET 平台的一个扩展，称为 Windows 身份验证基础（Windows Identity Foundation）。这个类模型大部分位于 `Microsoft.IdentityModel` 命名空间中。从 .NET 框架 4.5 开始，这个类模型成为 .NET 框架的重要组成部分，并移植到到 `System` 命名空间中，使用原有类模型的代码将无法运行。

在分布式系统中，身份凭证的提供者，可能并不是对这些感兴趣的一方。在这些情况下，我们需要对二者加以区分：

- 凭证提供方（identity provider）是一个实体，发布关于主体的凭证声明，这个声明通常包含该实体有权发布或验证过的信息；
- 凭证信赖方（relying party）是使用凭证提供方发布的凭证声明的实体（即：使用方或依赖方）。

图 15-7 展示了凭证提供方和信赖方之间的关系。分布式身份验证协议（如 WS-Federation）的任务如下：

- 为凭证信赖方提供请求机制，向凭证提供方请求获得关于一个主体的身份声明；
- 为凭证提供方提供发出声明的机制，并使发出请求的信赖方能够获得这些声明。

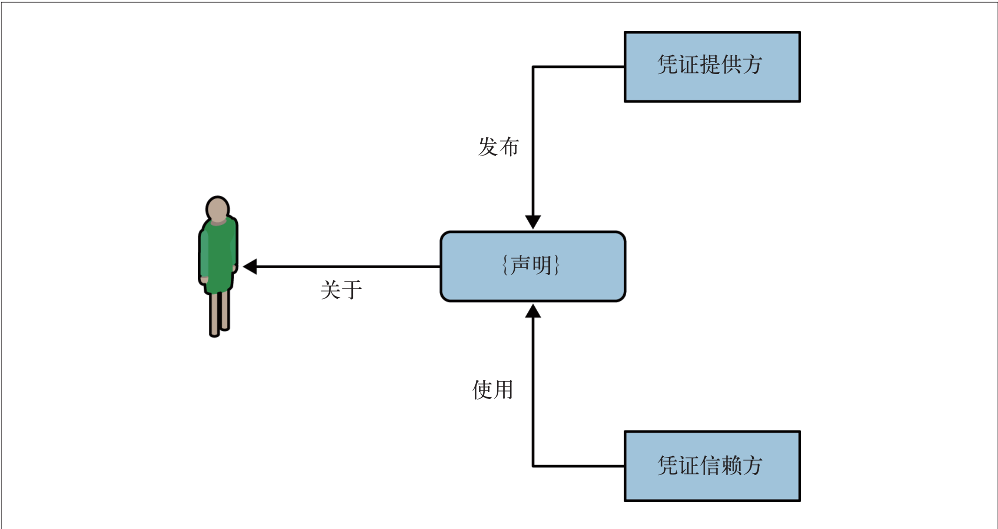


图 15-7：凭证提供方和凭证信赖方

15.3.2 获取和设置当前用户对象

.NET 框架早期的版本通过静态属性 `Thread.CurrentPrincipal` 进行当前用户对象的获取和设置。但是，这一技术现在存在两个问题。第一，一个请求不一定是由单个线程执行的。特别是异步编程模型的广泛使用，使得一个请求不再和单个执行线程关联在一起。第二，一些 .NET 框架组件，例如 ASP.NET 和 WCF，定义了访问和定义当前用户身份信息的其他途径。例如，在 ASP.NET 中，`HttpContext` 类有一个静态属性 `User`，其类型为 `IPrincipal`。用户身份信息在多处保存，增加了信息不一致的可能性。

在 ASP.NET Web API 1.0 中，托管方式决定了应该如何在消息处理程序管道中设置当前用户对象。如果你使用的是自托管，给 `Thread.CurrentPrincipal` 赋值即可。但是，如果使用的是 Web 托管，你必须给 `Thread.CurrentPrincipal` 和 `HttpContext.Current.User` 都赋值。一个常用的技巧是检查 `HttpContext.Current` 不为 `null`：

```
Thread.CurrentPrincipal = principalToAssign;
if (HttpContext.Current != null)
{
    HttpContext.Current.User = principalToAssign;
}
```

可是，如果使用这种技术，即便是采用自托管方式，也会造成应用程序对 `System.Web` 程序集的依赖。

在 ASP.NET Web API 2.0 中，你可以使用新的 `HttpRequestContext` 类，解决这一问题。首先，你必须获得当前用户对象，将其赋给当前的请求对象，而不是设置静态属性。其次，不同的托管方式可以使用不同的 `HttpRequestContext` 实现：

- 自托管使用 `SelfHostHttpRequestContext`，直接给 `Thread.CurrentPrincipal` 属性赋值；
- Web 托管使用 `WebHostHttpRequestContext`，给 `Thread.CurrentPrincipal` 和 `HttpContext.Current.User` 属性都赋值；
- OWIN 托管使用 `OwinHttpRequestContext`，给 `Thread.CurrentPrincipal` 和当前的 OWIN 上下文对象赋值。

遗憾的是，Web API 的这两个版本没有通用的方法。在这本书的后面部分，我们将主要使用 Web API 2.0 版本的方法。

15.3.3 基于传输的身份验证

我们前面介绍过，TLS 协议也可以用于身份认证，为传输方提供远端对等方的身份标识。在接下来的小节中，我们将介绍如何使用 TLS 的这个功能，获取服务器和客户端身份验证。

15.3.4 服务器身份验证

当客户端使用 https 请求 URI 发送 HTTP 请求时，所用的连接必须始终受到 TLS 或 SSL 的保护，以确保发送消息的完整性和保密性。除此之外，客户端还必须在握手协商中获得服务器证书，将其中的标识与 URI 主机名进行比较，以检查服务器身份。这种验证能确保 HTTP 请求消息只发送给通过身份验证的服务器，解决了服务器身份验证问题。

从证书中获得服务器标识的方法如下：

- 如果证书包含一个类型为 DNS 名的*主体别名*（subject alternative name）扩展，那么就使用这个值；
- 否则，就使用证书的主体字段中的*常用名*（common name）。

如果*主体别名*扩展包含多个名字，那么 URI 主机名可以匹配其中任何一个。有了这个功能，我们可以对不同主机名（例如：www.example.net 和 api.example.net）使用同一个证书，当这些主机名都绑定到同一个 IP 地址时会十分有用。例如：在 IIS 7.5 之前的版本中，使用同一个 IP 地址和端口的不同 https 绑定都必须使用同一个证书。

服务器证书中的名字也可以包含通配符（例如：*.example.net）。举个例子，*.example 匹配主机名 www.example.net。如果服务器有多个租户，在服务器证书颁发时还无法知道主机名，就可以在名字中使用通配符。例如：Azure Service Bus 目前使用的证书包含两个别名：*.servicebus.windows.net 和 servicebus.windows.net，可以匹配如 mytenant-name.servicebus.windows.com 这样的主机名。

目前，基于 TLS 的服务器身份验证使用 PKI 信任模式（详见附录 G），在这种模式中，整体安全依赖于一组认证机构的正确行为。遗憾的是，PKI 信任模式很容易受到 MITM（Man-In-The-Middle，中间人）攻击。例如，如果攻击者控制了 CA（Certificate Authority，认证授权机构）的名字验证程序，就可以将攻击者控制的公用密钥与他人的名字进行绑定，伪造证书。如果攻击者能够使用 CA 的密钥颁发伪造的证书，也会导致同样的后果。

一些平台在默认情况下配置了很多受信任的根证书颁发机构（root certification authority），使这一问题更加严重。例如，Mozilla 项目（即：Fixfox 浏览器）使用的根证书列表有 150 多个不同的条目（参见 https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/）。请注意，如果攻击者控制了这些证书颁发机构中的任何一个，那么就可以对任何服务器发起 MITM 攻击，即便该服务器的证书不是由受控制的机构颁发的，也不能幸免。

要解决这个安全问题，一个方法是在服务器证书验证过程中加入额外的上下文要求。其中一个额外要求称为*证书识别*（certificate pinning），将链中的证书与一组固定的已知证书进行比较，这组已知证书称为*锁定证书集合*（pinset）。如果客户端与服务器的第一个交互确定没有受到 MITM 攻击，服务器提供的证书链就可以用于构建锁定证书集合。选择这种方式，是因为服务器改变其使用的根证书的概率很低。

另一个方法是使用一个静态的基于上下文的锁定证书集合。例如，Chromium 浏览器（参见 http://blog.chromium.org/2011/06/new-chromium-security-features-june.html）对用户连接到 Gmail 和谷歌账号服务器时使用的 CA 进行了限制。Twitter API 安全最佳实践（参见 https://dev.twitter.com/overview/api/ssl）是使用这种方法的另一个例子，其中声称任何客户端应用程序都应该确保 Twitter 服务器返回的证书链包含一部分已核准的 CA。

在使用 `HttpClient` 时，你可以使用 `WebRequestHandler` 客户端处理程序和一个定制的证书验证回调程序，强制进行证书识别，代码如下所示：

```
private readonly CertThumbprintSet verisignCerts = new CertThumbprintSet(
    "85371ca6e550143dce2803471bde3a09e8f8770f",
    "62f3c89771da4ce01a91fc13e02b6057b4547a1d",

    "4eb6d578499b1ccf5f581ead56be3d9b6744a5e5",
    "5deb8f339e264c19f6686f5f8f32b54a4c46b476"
    );

[Fact]
public async Task Twitter_cert_pinning()
{
    var wrh = new WebRequestHandler();
    wrh.ServerCertificateValidationCallback =
        (sender, certificate, chain, errors) =>
        {
            var caCerts = chain.ChainElements
                .Cast<X509ChainElement>().Skip(1)
                .Select(elem => elem.Certificate);

            return errors == SslPolicyErrors.None &&
                caCerts.Any(cert =>
                 verisignCerts.Contains(cert.GetCertHashString()));
        };

    using (var client = new HttpClient(wrh))
    {
        await client.GetAsync("https://api.twitter.com");

        var exc = Assert.Throws<AggregateException>(() =>

            client.GetAsync("https://api.github.com/").Result);
        Assert.IsType<HttpRequestException>(exc.InnerExceptions[0]);
    }
}
```

`verisignCerts` 中包含了 pinset，是一组保存在定制的 `CertThumbprintSet` 类中的证书指纹，`CertThumbprintSet` 定义如下：

```
public class CertThumbprintSet : HashSet<string>
{
    public CertThumbprintSet(params string[] thumbs)
        :base(thumbs, StringComparer.OrdinalIgnoreCase)
    {}
}
```

示例中使用的 HttpClient 以一个显示实例化的 WebRequestHandler 为参数进行创建。这个处理程序提供了 ServerCertificateValidationCallback 属性，可以定义为一个委托方法，由运行时在标准的内建证书验证过程结束后调用。这个委托方法接收内建过程的验证结果，其中包含错误发生的信息，并返回一个布尔值，表示最终的验证结果。ServerCertificateValidationCallback 属性可以用于重写内建的验证结果，或者执行附加的验证步骤。

在示例中，我们使用这个属性执行附加的验证步骤。只有满足如下条件时服务器证书才认为是有效的：

- 内建验证成功，即 errors == SslPolicyErrors.None；
- 证书链至少包含一个已知的 CA 证书（锁定的证书）。

请注意，证书链的检查中跳过了叶证书，因为我们只需要确保 CA 证书属于一个已知的锁定证书集合即可。还要注意的是，这个锁定证书集合只适用于 Twitter 上下文。正如 Asset.Throws 所示，如果使用这一配置连接其他的服务器（api.github.com），会导致证书验证异常。

在这本书编写时，证书识别的使用还是高度依赖于上下文，通常必须与管理服务器的机构进行协商。Twitter 安全最佳实践就是使用锁定证书集合策略的一个例子。但是，人们正在制定一些新的规范，尝试将这一技术变得更为通用。其中一个规范称为 HTTP 公共密钥锁定扩展（Publick Key Pinning Extension for HTTP），这一规范允许服务器告诉客户端在一段指定时间内锁定其提供的证书，具体实现方法是在响应标头中加入锁定的证书以及锁定时间段：

```
Public-Key-Pins: pin-sha1="4n972HfV354KP560yw4uqe/baXc=";
        pin-sha1="qvTGHdzF6KLavt4PO0gs2a6pQ00=";
        pin-sha256="LPJNul+wow4m6DsqxbninhsWHlwfp0JecwQzYpOLmCQ=";
        max-age=2592000
```

对服务器进行身份验证时，你应该强制实现的另一个方面是，确保现有的证书没有撤销。但是，要实现这一功能，ServicePointManager 必须通过静态属性 CheckCertificateRevocationList，明确进行配置：

```
ServicePointManager.CheckCertificateRevocationList = true;
```

.NET HTTP 客户端基础结构使用 ServicePointManager 类，通过 ServicePoint 对象，获得到服务器的连接。

我们也可以使用 WebRequestHandler.ServerCertificateValidationCallback，确保执行适当的撤销验证。

```
return errors == SslPolicyErrors.None &&
    caCerts.Any(cert => verisignCerts.Contains(cert.GetCertHashString())) &&
    chain.ChainPolicy.RevocationMode == X509RevocationMode.Online;
```

以上代码中的最后一个条件，使用了回调委托方法接收的 `X509Chain` 的 `ChainPolicy` 属性，确保撤销验证使用的是在线机制模式。如果这个条件不为真，那么代码就不接受这个服务器证书，并抛出一个异常。

15.3.5 客户端身份验证

TLS 传输安全机制也可以提供客户端身份验证，但要使用*客户端证书*，增加了客户端的复杂度和基础结构要求，客户端必须存储私有密钥，并需要向客户端颁发证书。因为这些条件要求，人们通常不会进行 TLS 客户端身份验证。然而，在以下场景中你应该认真考虑使用 TLS 客户端身份验证：

- 系统安全需要使用可靠性更高的客户端身份验证方式；
- 系统已经具有 PKI（Public Key Infrastructure，公钥基础设施），可用于颁发客户端证书。

例如，多个欧洲国家正在推行电子身份证计划（electronic identity initiatives，参见 https://www.eid-stork.eu），让每个公民拥有一张包含个人证书（以及相关私钥）的智能卡。这些证书可以用于对公民与电子政务网站的 TLS 交互进行身份验证。因此，如果在这些国家开发电子政务 Web API，你就应该考虑使用 TLS 客户端身份验证。目前，实现 TLS 客户端身份验证的主要限制在于，难以在便携设备上使用这些只能卡，例如：智能手机或平板电脑。

Windows Azure Service Management REST API 是一个使用基于 TLS 的客户端身份验证的公共 Web API 的一个具体例子。客户端请求必须使用一个*管理证书*，该证书预先与管理服务进行了关联。在这个 API 中，客户端自己生成证书，不需要具有公钥基础设施，从而简化了证书的使用。

如果在 IIS 上进行 Web API 托管，你可以在 IIS 管理器的功能视图中，选择 SSL 设置，配置 TLS 客户端身份验证，如图 15-8 所示。

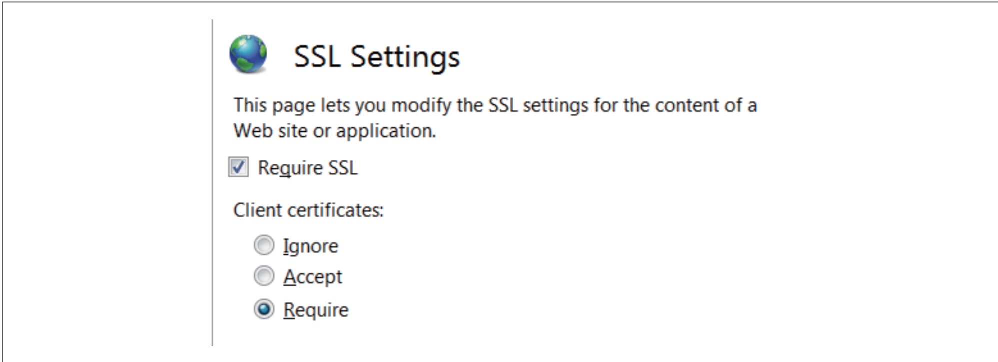


图 15-8：配置 TLS 客户端身份验证

这个设置对一个文件夹有效，为 TLS 握手提供三个选项：

- 不请求客户端证书（Ignore）；
- 请求客户端证书，但允许客户端不发送证书（Accept）；
- 请求客户端证书，并且要求客户端发送证书（Require）。

使用自托管时，你可以使用 `netsh` 命令行工具，在设置服务器 TLS 证书时指定 `clientcertnegotiation` 参数，配置客户端身份验证。

```
netsh http add sslcert (...) clientcertnegotiation=enable
```

客户证书并不包含在客户端发送的 HTTP 请求中，而是请求使用的传输连接的一个属性。这个证书通过 Web API 托管层与请求对象进行关联。对于自托管，你必须进行一个额外的配置，将配置对象的 `ClientCredentialType` 属性设置为 `Certificate`。

```
var config = new HttpSelfHostConfiguration("https://www.example.net:8443");
config.ClientCredentialType = HttpClientCredentialType.Certificate;
```

进行这项配置，证书信息才能从自托管的 WCF 适配器向上传到请求对象。这个配置并不影响 TLS 连接的协商和建立——也就是说，这个配置并不能替代基于 `netsh` 的配置工作。如果你使用的是 Web 托管，那么就不需要进行这项配置。

`HttpSelfHostConfiguration` 也提供了属性 `X509CertificateValidator`，以定义附加的定制证书验证过程。请注意，`X509CertificateValidator` 属性不会改变 TLS 的 HTTP.SYS 实现的证书验证，只是增加了另一个验证过程。而且，只有使用自托管时，才能使用 `X509CertificateValidator` 属性。

使用基于 TLS 的客户端身份验证时，无论使用何种托管方式，你都需要在服务器端检查协商证书，以获得客户端的身份。你可以使用 `GetClientCertificate` 扩展方法，从请求消息中得到客户端证书，如示例 15-1 所示。

示例 15-1：访问客户端证书

```
public class HelloController : ApiController
{
    public HttpResponseMessage Get()
    {
        var clientCert = Request.GetClientCertificate();
        var clientName = clientCert == null ? "stranger" : clientCert.Subject;
        return new HttpResponseMessage
        {
            Content = new StringContent("Hello there, " + clientName)
        };
    }
}
```

ASP.NET Web API 2.0 中，你也可以使用新的 `HttpRequestContext` 类，取得客户端证书：

```
var clientCert = Request.GetRequestContext().ClientCertificate;
```

获取客户端证书，还有一个更好的方法，是使用如示例 15-2 中的消息处理程序。示例 15-2 中的消息处理程序将接收到的客户端证书映射到一个基于声明的用户标识。使用这种方式，不管使用何种身份验证机制，我们都可以得到统一格式的用户标识。我们也可以使用这个消息处理程序，执行附加的证书验证。在默认情况下，TLS 的 HTTPS.SYS 实现会使用 Windows Store 中存在的任何受信任的根证书颁发机构，进行证书验证。但是，我们也许希望对证书颁发机构进行一些限制。

示例 15-2：进行证书验证和声明映射的消息处理程序

```
public class X509CertificateMessageHandler : DelegatingHandler
{
    private readonly X509CertificateValidator _validator;
    private readonly Func<X509Certificate2, string> _issuerMapper;
    const string X509AuthnMethod =
       "http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/x509";

    public X509CertificateMessageHandler(
        X509CertificateValidator validator,
        Func<X509Certificate2,string> issuerMapper)
    {
        _validator = validator;
        _issuerMapper = issuerMapper;
    }

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        var cert = request.GetClientCertificate();
        if (cert == null) return await base.SendAsync(request, cancellationToken);
        try
        {
            _validator.Validate(cert);
        }
        catch (SecurityTokenValidationException)
        {
            return new HttpResponseMessage(HttpStatusCode.Unauthorized);
        }
        var issuer = _issuerMapper(cert);
        if (issuer == null)
        {
            return new HttpResponseMessage(HttpStatusCode.Unauthorized);
        }

        var claims = ExtractClaims(cert, issuer);
        var identity = new ClaimsIdentity(claims, X509AuthnMethod);
        AddIdentityToCurrentPrincipal(identity, request);

        return await base.SendAsync(request, cancellationToken);
    }
```

```
    private static IEnumerable<Claim> ExtractClaims(
        X509Certificate2 cert,
        string issuer)
    {
        ...
    }

    private static void AddIdentityToCurrentPrincipal(ClaimsIdentity identity)
    {
        ...
    }
}
```

首先，这个消息处理程序使用 GetClientCertificate 扩展方法，从请求消息中获得客户端证书，然后使用构造函数中定义的验证程序，执行附加的证书验证。X509CertificateValidator 是 .NET 框架提供的一个抽象基类，表示一个证书验证过程，其中定义了一组静态方法，进行常见的验证操作。

```
public abstract class X509CertificateValidator : ICustomIdentityConfiguration
{
       // 省略类成员及实现
       public static X509CertificateValidator None {get{...}}
       public static X509CertificateValidator PeerTrust {get{...}}
       public static X509CertificateValidator ChainTrust {get{...}}
       public static X509CertificateValidator PeerOrChainTrust{get{...}}
}
```

如果证书通过了附加的验证，那么消息处理程序就使用 Func<X509Certificate2, string>，获得证书颁发者的名称，用于创建获取的声明。获得证书颁发者的名称常用的两个策略为：

- 使用证书颁发者的名称（例如：CN=Demo Certification Authority, O=Web API Book）；
- 使用预先定义的注册表，将颁发客户端证书的 CA 证书映射到一个证书颁发者字符串。

.NET 框架提供一个 IssuerNameRegistry 类，实现上面列出的第二种策略。

获得证书颁发者名称之后，消息处理程序从客户端证书计算出表示请求者的声明集合。

```
private static IEnumerable<Claim> ExtractClaims(
    X509Certificate2 cert,
    string issuer)
{
    var claims = new Collection<Claim>
        {
            new Claim(ClaimTypes.Thumbprint,
                        Convert.ToBase64String(cert.GetCertHash()),
                    ClaimValueTypes.Base64Binary, issuer),
            new Claim(ClaimTypes.X500DistinguishedName,
                cert.SubjectName.Name,
                ClaimValueTypes.String, issuer),
            new Claim(ClaimTypes.SerialNumber, cert.SerialNumber,
                ClaimValueTypes.String, issuer),
```

```
            new Claim(ClaimTypes.AuthenticationMethod, X509AuthnMethod,
                ClaimValueTypes.String, issuer)
        };
    var email = cert.GetNameInfo(X509NameType.EmailName, false);
    if (email != null)
    {
        claims.Add(new Claim(ClaimTypes.Email, email,
            ClaimValueTypes.String, issuer));
    }
    return claims;
}
```

在这段代码中，我们将多个证书字段各自映射到单独的声明对象，即：证书散列指纹、主体名称、证书序列号以及主体的邮件地址（如果存在的话），还添加了一个表明身份验证方法的声明对象。

最后，消息处理程序创建了一个声明标识，将其加入当前的基于声明的用户对象。如果当前用户对象不存在，就创建一个新的当前用户对象。

```
private static void AddIdentityToCurrentPrincipal(ClaimsIdentity identity)
{
    private void AddIdentityToCurrentPrincipal(
        ClaimsIdentity identity,
        HttpRequestMessage request)
    {
        var principal = request.GetRequestContext().Principal as ClaimsPrincipal;
        if (principal == null)
        {
            principal = new ClaimsPrincipal(identity);
            request.GetRequestContext().Principal = principal;
        }
        else
        {
            principal.AddIdentity(identity);
        }
    }
}
```

使用消息处理程序，将客户端证书映射为一个基于声明的用户身份，下游的 Web API 运行时组件就可以总是以不变的方式使用用户身份信息。例如，前面的示例 15-1 中的 `HelloController` 现在可以重新进行实现，代码如下：

```
public class HelloController : ApiController
{
    public HttpResponseMessage Get()
    {
        var principal = User as ClaimsPrincipal;
        var name = principal
            .Identities.SelectMany(ident => ident.Claims)
            .FirstOrDefault(c => c.Type == ClaimTypes.Email).Value ?? "stranger";
        return new HttpResponseMessage
        {
```

```
            Content = new StringContent("Hello there, " + name)
        };
    }
}
```

与 ASP.NET MVC 的做法相似，ApiController 类提供一个属性 User，保存请求者的用户对象。还需要注意的是，使用用户身份的代码只需要处理声明，不必依赖基于传输的客户端机制。举个例子，如果客户端使用了新的身份验证机制，例如下一节中将要介绍的身份验证机制，消息处理程序的操作代码也无需进行修改。但是，在接着介绍基于消息的身份验证之前，我们必须了解客户端如何使用基于传输的客户端身份验证。

在客户端一方，如果使用基于 TLS 的身份验证配置，你必须直接处理第 14 章中介绍的 HttpClient 处理程序之一：HttpClientHandler 或 WebRequestHandler。

如果使用第一个选项，你需要明确配置 HttpClient 使用 HttpClientHanlder 实例，并将 HttpClientHanlder 的属性 ClientCertificateOptions 设置为 Automatic。

```
var client = new HttpClient(
    new HttpClientHandler{
        ClientCertificateOptions = ClientCertificateOption.Automatic
    });
// ...
```

由此生成的 HttpClient 接着就可以正常使用：如果在连接握手过程中，服务器要求客户端提供证书，HttpClientHandler 实例就会自动选择一个兼容的客户端证书。Windows Store 应用程序只能使用 HttpClientHandler。

对于经典场景（例如：控制台程序、WinForm 程序或 WPF 应用程序），我们还有第二个选项：使用 WebRequestHandler。

```
var clientHandler = new WebRequestHandler()
clientHandler.ClientCertificates.Add(cert);
var client = new HttpClient(clientHandler)
```

在这段代码中，cert 是一个 X509Certificate2 实例，表示客户端证书。这个 cert 实例可以直接从一个 PFX 文件构建或者从 Windows 的证书库中得到。

```
X509Store store = null;
try
{
    store = new X509Store(StoreName.My, StoreLocation.CurrentUser);
    store.Open(OpenFlags.OpenExistingOnly | OpenFlags.ReadOnly);
    // 从 store.Certificates 选择证书……
}
finally
{
    if(store != null) store.Close();
}
```

在讨论了如何使用传输安全机制提供客户端和服务器身份验证之后，我们现在要了解如何在 HTTP 消息层解决这些安全需求。

15.3.6 HTTP身份验证框架

我们在第 1 章中介绍过，HTTP 协议规范定义了一个通用的*身份验证框架*，在这个框架之上可以定义具体的身份验证机制。RFC 2617 中定义的*基础认证*和*摘要认证*方案就属于这种身份验证机制。HTTP 身份验证框架定义了响应状态码和消息标头，以及一个质询 - 响应序列，供具体的身份验证机制使用（参见第 1 章和附录 E）。

基础认证方案使用一个简单的用户名和密码对，对客户端进行身份验证。这些身份信息以如下方式加入请求消息中：

(1) 连接用户名和密码，使用 :（冒号）分隔符；
(2) 使用 Base64 对连接串进行编码，生成的字符串放置在 `Authorization` 标头的方案标识符 `Basic` 之后。

示例 15-3 中的代码使用基础认证从 GitHub API 获得用户信息。请注意，代码给请求消息加入了 `Authorization` 标头。

示例 15-3：使用基础认证从 GitHub 获得用户信息

```
using (var client = new HttpClient())
{
    var req = new HttpRequestMessage(
        HttpMethod.Get, "https://api.github.com/user");
    req.Headers.UserAgent.Add(new ProductInfoHeaderValue("webapibook","1.0"));
    req.Headers.Authorization = new AuthenticationHeaderValue(
        "Basic",
        Convert.ToBase64String(
            Encoding.ASCII.GetBytes(username + ':' + password))
        );
    var resp = await client.SendAsync(req);
    Console.WriteLine(resp.StatusCode);
    var cont = await resp.Content.ReadAsStringAsync();
    Console.WriteLine(cont);
}
```

在服务器端，我们可以使用如示例 15-4 中的消息处理程序，强制使用基础认证。示例 15-4 中的消息处理程序检查请求消息中是否存在带有 `Basic` 方案的 `Authorization` 标头，并尝试获取用户名和密码，进行验证。如果验证通过，消息处理程序就生成一个描述请求者的用户对象，将其添加到请求上下文，然后将消息处理委托给下一个处理程序。如果这些条件中任何一个失败，那么消息处理程序会产生一个状态码为 `401` 的响应消息，终止请求的处理。

如果响应消息的状态码为 `401`，那么处理程序会给响应消息加入一个 `WWW-Authenticate` 标

头，其中包含所需的身份验证方案和域名。

示例 15-4：基础认证消息处理程序

```
public class BasicAuthenticationDelegatingHandler : DelegatingHandler
{
    private readonly Func<string, string, Task<ClaimsPrincipal>> _validator;
    private readonly string _realm;

    public BasicAuthenticationDelegatingHandler(string realm, Func<string, string,
        Task<ClaimsPrincipal>> validator)
    {
        _validator = validator;
        _realm = "realm=" + realm;
    }

    protected async override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        HttpResponseMessage res;
        if (!request.HasAuthorizationHeaderWithBasicScheme())
        {
            res = await base.SendAsync(request, cancellationToken);
        }
        else
        {
            var principal = await
                request.TryGetPrincipalFromBasicCredentialsUsing(_validator);
            if (principal != null)
            {
                request.GetRequestContext().Principal = principal;
                res = await base.SendAsync(request, cancellationToken);
            }
            else
            {
                res = request.CreateResponse(HttpStatusCode.Unauthorized);
            }
        }

        if (res.StatusCode == HttpStatusCode.Unauthorized)
        {
            res.Headers.WwwAuthenticate.Add(
                new AuthenticationHeaderValue("Basic", _realm));
        }
        return res;
    }
}
```

如果身份验证通过，那么产生的用户对象就通过 `HttpRequestContext.Principal` 属性，添加到请求上下文。用户对象的获取和验证是通过一个扩展方法完成的，该方法定义如下：

```
public static async Task<ClaimsPrincipal>
    TryGetPrincipalFromBasicCredentialsUsing(
```

```
        this HttpRequestMessage req,
        Func<string,string,Task<ClaimsPrincipal>> validate)
{
    string pair;
    try
    {
        pair = Encoding.UTF8.GetString(
            Convert.FromBase64String(req.Headers.Authorization.Parameter));
    }
    catch (FormatException)
    {
        return null;
    }
    catch (ArgumentException)
    {
        return null;
    }
    var ix = pair.IndexOf(':');
    if (ix == -1) return null;
    var username = pair.Substring(0, ix);
    var pw = pair.Substring(ix + 1);
    return await validate(username, pw);
}
```

这个方法使用构造函数中传入的委托方法进行用户名和密码验证，与具体验证逻辑无关。

15.3.7 实现基于HTTP的身份验证

在前面的示例中，我们使用 Web API 消息处理程序，实现了服务器端的 HTTP 身份验证。然而，我们还可以使用别的架构选项：使用 Web API 筛选器，在管道中实现身份验证；或者在托管层进行身份验证。接下来我们将讨论不同选项的利弊。

如果在 Web API 筛选器上实现身份验证，你可以访问更多的请求信息，其中有：

- 所选的控制器和路由；
- 路由参数；
- 操作参数（如果使用操作筛选器）。

如果你的身份验证逻辑依赖这些信息，就可以使用 Web API 筛选器实现身份验证。而且，Web API 筛选器还可以选择性地应用于一部分控制器或者操作。

但是，在筛选器层进行身份验证也有一些很严重的缺点。

- 在管道后端才能检测到未经认证的请求，增加了拒绝请求的计算开销。
- 在管道后端才能使用请求者的身份信息。这意味着其他的中间层组件，如缓存中间件，可能无法访问这些身份信息。如果使用私有缓存（即：按用户进行缓存），那么这种方式会产生严重的限制。

另外一个选项是，在消息处理程序层实现身份验证功能，我们之前就是使用的这种方式。消息处理程序在托管适配层之后立刻运行，因此拒绝请求的开销会比较小。而且，之后的所有处理程序都可以使用请求者的身份信息。如果你使用 `HttpClient`，消息处理程序在客户端也可使用，因此这种方式使得系统呈现一种有趣的对称性。

然而，OWIN 规范（参见第 11 章）的使用，引入了实现身份验证的另一个选项：OWIN 中间件。这种方式有一些重要的优点。

- 扩展了使用范围。同一个身份验证中间件可以由多个框架使用，不仅仅限于 ASP.NET Web API。
- 下游的 OWIN 中间件，例如：缓存或日志，立即可以使用请求者的身份信息。

事实上，OWIN 规范的引入意味着，所有不与框架相关的中间层都最好实现为 OWIN 中间件。Katana 项目遵循的就是这种方式，提供了一组身份验证中间件的实现。

显然，只有在 OWIN 服务器上进行 Web API 托管时，你才能够使用这种方式。但是，随着人们越来越多地采纳 OWIN 规范，这种方式也会变得更为普遍。

15.3.8　Katana身份验证中间件

Katana 项目 2.0 包含一组中间件实现，提供用于不同范围的多种身份验证机制，从传统的基于 cookie 的身份验证到基于 OAuth 2.0 的身份验证。这些中间件的实现基于一个可扩展的类结构，我们接下来要对进行介绍。

这个类结构的根是基础抽象类 `AuthenticationMiddleware<Toption>`（参见示例 15-5），具体的身份验证中间件可以对这个基类进行派生。

示例 15-5：身份验证中间件基础类

```
public abstract class AuthenticationMiddleware<TOptions> : OwinMiddleware
        where TOptions : AuthenticationOptions
{
    protected AuthenticationMiddleware(OwinMiddleware next, TOptions options)
        : base(next)
    { ... }

    public TOptions Options { get; set; }

    public override async Task Invoke(IOwinContext context)
    {
        AuthenticationHandler<TOptions> handler = CreateHandler();
        await handler.Initialize(Options, context);
        if (!await handler.InvokeAsync())
        {
            await Next.Invoke(context);
        }
        await handler.TeardownAsync();
```

```
    }

    protected abstract AuthenticationHandler<TOptions> CreateHandler();
}
```

这个类有一个类型参数 TOption，定义了身份验证中间件的配置选项，如身份验证。通常情况下，开发定制的身份验证中间件需要定义一个具体的选项类，示例 15-6 就定义了这样一个选项类。

示例 15-6：基础认证选项

```
public class BasicAuthenticationOptions : AuthenticationOptions
{
    public Func<string, string, Task<AuthenticationTicket>>
        ValidateCredentials { get; private set; }
    public string Realm { get; private set; }

    public BasicAuthenticationOptions(
        string realm,
        Func<string, string, Task<AuthenticationTicket>> validateCredentials)
        : base("Basic")
    {
        Realm = realm;
        ValidateCredentials = validateCredentials;
    }
}
```

当中间件管道处理请求时，会调用 OwinMiddleware.Invoke 方法，将身份验证操作委托给由 CreateHandler 方法提供的一个*身份验证处理程序*实例。因此，定制身份验证中间件的主要任务通常就是定义这个 CreateHandler 方法（参见示例 15-7）。

示例 15-7：基础认证中间件

```
class BasicAuthnMiddleware : AuthenticationMiddleware<BasicAuthenticationOptions>
{
    public BasicAuthnMiddleware(
        OwinMiddleware next,
        BasicAuthenticationOptions options)
        : base(next, options)
    {}

    protected override AuthenticationHandler<BasicAuthenticationOptions>
        CreateHandler()
    {
        return new BasicAuthenticationHandler(Options);
    }
}
```

了解基础中间件如何使用这个处理程序，可以帮助你更好地理解处理程序的工作。如示例 15-5 所示，基础中间件的 Invoke 方法在创建处理程序之后，执行了三项任务。首先，Invoke 方法调用了处理程序的 Initialize 方法。我们稍后会看到，处理程序的这

个 Initialize 方法触发了大部分的身份验证操作。随后，Invoke 方法调用了处理程序的 InvokeAsync 方法。如果 InvokeAsync 方法返回 false，那么 Invoke 方法会调用下一个中间件，否则 Invoke 方法会终止请求处理，也就是说不再调用下游的中间件。最后，在 Invoke 方法调用 TeardownAsync 方法，释放处理程序实例。请注意，中间件和处理程序对象的生存期不同：中间件存在于整个应用程序过程中，而处理程序实例只存在于单个请求处理过程。这也是中间件和处理程序程序作为两个单独概念存在的原因之一。

抽象类 AuthenticationHanlder 提供了大部分通用的的身份验证协调功能，具体的身份验证逻辑则委托给由具体的派生类实现的挂钩方法。对于基于 HTTP 框架的身份验证，有两个挂钩方法尤为重要：AuthenticateCoreAsync 和 ApplyResponseChallengeAsync。

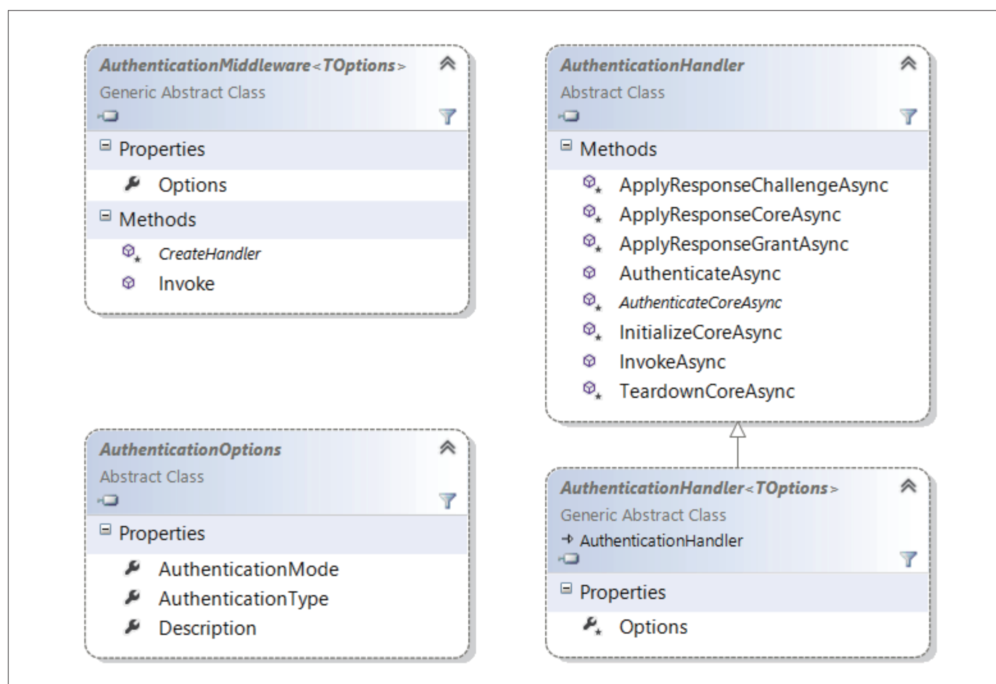


图 15-9：Katana 身份验证中间件

AuthenticateCoreAsync 方法由处理程序的 Initialize 方法调用，对当前请求进行身份验证。示例 15-8 中实现的 AuthenticateCoreAsync 方法，尝试从请求的 Authorization 标头获得 Basic 身份信息进行验证。如果验证通过，AuthenticateCoreAsync 方法返回一个用户身份，由基类的 Initialize 方法添加到请求——具体地说，是添加到上下文条目 server.User。如果验证失败，AuthenticateCoreAsync 方法返回 null，说明身份验证没有通过。

处理程序的 Initialize 方法将 ApplyResponseChallengeAsync 方法注册到响应事件 OnSendingHeaders，这一事件在响应的标头开始发送之前触发。示例 15-8 中实现的 Apply

ResponseChallengeAsync，在响应状态码为 401 时，给响应添加一个 WWW-Authenticatae 质询标头，其中包含 Basic 认证方案。

示例 15-8：基础认证处理程序

```
class BasicAuthenticationHandler : AuthenticationHandler<BasicAuthenticationOptions>
{
    private readonly string _challenge;

    public BasicAuthenticationHandler(BasicAuthenticationOptions options)
    {
        _challenge = "Basic realm=" + options.Realm;
    }

    protected override async Task<AuthenticationTicket> AuthenticateCoreAsync()
    {
        var authzValue = Request.Headers.Get("Authorization");
        if (string.IsNullOrEmpty(authzValue) || !authzValue.StartsWith("Basic ",
            StringComparison.OrdinalIgnoreCase))
        {
            return null;
        }
        var token = authzValue.Substring("Basic ".Length).Trim();
        return await
            token.TryGetPrincipalFromBasicCredentialsUsing(
                Options.ValidateCredentials);
    }

    protected override Task ApplyResponseChallengeAsync()
    {
        if (Response.StatusCode == 401)
        {
            var challenge = Helper.LookupChallenge(
                Options.AuthenticationType, Options.AuthenticationMode);
            if (challenge != null)
            {
                Response.Headers.AppendValues("WWW-Authenticate", _challenge);
            }
        }
        return Task.FromResult<object>(null); }
    }
}
```

异步挂钩方法 AuthenciationTicket 返回值为 AuthenticationTicket 类型，这是 Katana 项目进入的一个新类型，用于表示用户身份。AuthenticationTicket 类包含一个声明身份和一组身份验证属性。

```
public class AuthenticationTicket
{
    public AuthenticationTicket(
        ClaimsIdentity identity, AuthenticationProperties properties)
    {
        Identity = identity;
        Properties = properties;
```

```
    }

    public ClaimsIdentity Identity { get; private set; }
    public AuthenticationProperties Properties { get; private set; }
}
```

为了实现中间件注册，定制的身份验证中间件通常也提供一个如示例 15-9 所示的扩展方法，以使用如下代码：

```
app.UseBasicAuthentication(
    new BasicAuthenticationOptions("webapibook", (un, pw) => {
    /* 身份验证逻辑 */
}));
```

示例 15-9：用于注册基础认证中间件的扩展方法

```
public static class BasicAuthnMiddlewareExtensions
{
    public static IAppBuilder UseBasicAuthentication(
        this IAppBuilder app, BasicAuthenticationOptions options)
    {
        return app.Use(typeof(BasicAuthnMiddleware), options);
    }
}
```

15.3.9 主动和被动的身份验证中间件

在 OWIN 规范中，中间件在请求到达 Web 框架之前对其进行访问。也就是说，在中间件运行时，可能还不知道具体的身份验证要求。假设我们有一个经典的 Web 应用程序以及一个托管在同一 OWIN 宿主上的 Web API。Web 应用程序可能使用 cookie 和基于表单的身份验证，而 Web API 可能使用基础认证。如果对 Web API 请求错误地使用了 cookie 进行身份验证，那么可能会导致安全问题，例如 CSRF（Cross-Site Request Forgery，跨站请求伪造），因为 cookie 是由基于浏览器的用户代理自动发送的。

因此，Katana 引入了主动和被动身份验证模式的概念。如果使用*主动*（active）模式，身份验证中间件会*主动地*尝试对请求进行身份验证，如果验证成功就将身份信息加入请求上下文。如果响应状态码为 `401`，身份验证中间件还会给响应加入质询信息。另一方面，处于*被动*（passive）模式的中间件只是将自己注册到一个*身份验证管理器*（authentication manager）。只有当明确得到调用时，身份验证处理程序才会尝试对请求进行身份验证并生成身份信息。只有当身份验证管理器对其进行明确调用时，处于被动模式的身份验证中间件才会给响应加入质询信息。

*身份验证管理器*也是 Katana 引入的一个新概念。身份验证管理器定义了一个接口，其他组件（如 Web 应用程序）可以通过这个接口于身份验证中间件进行交互。在下一节介绍 Web API 的身份验证筛选器时，我们会介绍使用身份验证管理器的具体例子。

属性 AuthenticationOptions.AuthenticationMode 定义了身份验证中间件的操作模式。身份验证中间件的大部分实现并不需要知道这个身份验证模式，主动和被动模式的行为差异是由基类 AuthenticationHandler 中的通用代码导致的。例如，只有当身份验证模式为主动模式时，基类才会调用 AuthenticateCoreAsync 方法。

这条规则有一个例外：给响应添加质询信息。不管配置的身份验证模式是什么，基础结构总是会调用 ApplyResponseChallengeAsync 方法。但是，只有当身份验证模式为主动模式，或者认证方案已加入身份验证管理器时，ApplyResponseChallengeAsync 方法才应该添加质询信息。

示例 15-8 中的 ApplyResponseChallengeAsync 方法使用了工具方法 Helper.LookupChallenge，以判断是否应该添加质询信息。

15.3.10 Web API身份验证筛选器

如前所述，我们可以选择将身份验证操作放在 Web API 筛选器中。Web API 2.0 进入了一个新的操作管道阶段，专门用于身份验证处理。这个阶段由身份验证筛选器组成，在授权筛选器阶段之前执行，也就是说，身份验证筛选器阶段是操作管道的第一个阶段。

身份验证筛选器接口包含两个异步方法，定义如下：

```
public interface IAuthenticationFilter : IFilter
{
    Task AuthenticateAsync(
        HttpAuthenticationContext context,
        CancellationToken cancellationToken);
    Task ChallengeAsync(
        HttpAuthenticationChallengeContext context,
        CancellationToken cancellationToken);
}
```

身份验证管道阶段分为两段：请求处理和响应处理。在请求处理中，Web API 运行时调用每个已注册的身份验证筛选器的 AuthenticateAsync 方法，传入身份验证上下文参数，其中包含了操作上下文和当前请求。

```
public class HttpAuthenticationContext
{
    public HttpActionContext ActionContext { get; private set; }
    public IPrincipal Principal { get; set; }
    public IHttpActionResult ErrorResult { get; set; }
    public HttpRequestMessage Request { get { ... }}
    ...
}
```

每个筛选器的 AuthenticateAsync 方法负责对上下文中的请求进行身份验证。如果请求消息中没有适当身份验证方案的身份信息，那么筛选器不会修改上下文。如果身份信息存在

而且有效，那么筛选器会将上下文的 Principal 属性设置为通过身份验证的用户对象；如果身份信息无效，那么上下文的 ErrorResult 属性会设置为一个状态码为 401 的响应消息，告诉运行时发生了一个身份验证错误，因此请求处理阶段会立即终止，运行时不再接着调用 AuthenticateAsync 方法，而是开始响应处理阶段。

如果没有筛选器对上下文的 ErrorResult 赋值，那么运行时会继续进行到下一个管道阶段，无论上下文的用户对象是否得到赋值，这就将是否授权匿名请求留给了上层决定。

当响应从上层返回，或者身份验证筛选器生成一个错误响应时，身份验证阶段的响应处理段就开始了。在这个响应段，运行时调用每个已登记的身份验证筛选器的 ChallengeAsync 方法，并传入一个*质询上下文*（challenge context）。

```
public class HttpAuthenticationContext
{
    public HttpActionContext ActionContext { get; private set; }
    public IPrincipal Principal { get; set; }
    public IHttpActionResult ErrorResult { get; set; }
    public HttpRequestMessage Request { get { ... }}
    // 省略成员和定义
}
```

身份验证筛选器可以借此机会检查结果信息，如果需要还可以加入认证质询信息。请注意，无论在身份验证阶段的请求处理段发生了什么，运行时总是会调用*所有*身份验证筛选器的 ChallengeAsync 方法 .

示例 15-10 实现了一个使用基础认证方案的身份验证筛选器。因为身份验证过程可能需要于外部系统进行通信（例如：用户身份数据库），我们使用了一个返回 Task<ClaimsPrincipal> 的函数。如果请求消息中不存在 Authorization 标头，或者认证方案不是 Basic，那么 AuthenticateAsync 方法不会修改上下文。如果用户身份信息存在但却无效，那么 AuthenticateAsync 方法会将 ErrorResult 赋值为 UnauthorizedResult，代表一个状态为 401 的响应。UnauthorizedResult 的质询列表为空，因为质询信息会由 ChallengeAsync 方法在响应处理阶段添加。

ChallengeAsync 只是检查响应状态是否为 401，如果是则添加合适的质询信息。我们必须使用帮助方法 ActionResultDelegate，因为上下文的 Response 属性的类型是 IHttpActionResult 接口，而不是直接的 HttpResponse 类。帮助方法 ActionResultDelegate 可以将一列 IHttpActionResult 实例组合成一个。

示例 15-10：基础认证消息处理程序

```
public class BasicAuthenticationFilter : IAuthenticationFilter
{
    private readonly Func<string, string, Task<ClaimsPrincipal>> _validator;
    private readonly string _realm;
    public bool AllowMultiple { get { return false; } }
```

```
        public BasicAuthenticationFilter(
            string realm, Func<string, string, Task<ClaimsPrincipal>> validator)
        {
            _validator = validator;
            _realm = "realm=" + realm;
        }

        public async Task AuthenticateAsync(
            HttpAuthenticationContext context,
            CancellationToken cancellationToken)
        {
            var req = context.Request;
            if (req.HasAuthorizationHeaderWithBasicScheme())
            {
                var principal = await
                    req.TryGetPrincipalFromBasicCredentialsUsing(_validator);
                if (principal != null)
                {
                    context.Principal = principal;
                }
                else
                {
                    // 质询信息将由 ChallengeAsync 添加
                    context.ErrorResult = new UnauthorizedResult(
                        new AuthenticationHeaderValue[0], context.Request);
                }
            }
        }

        public Task ChallengeAsync(
            HttpAuthenticationChallengeContext context,
            CancellationToken cancellationToken)
        {
           context.Result = new ActionResultDelegate(context.Result, async (ct, next) =>
           {
                var res = await next.ExecuteAsync(ct);
                if (res.StatusCode == HttpStatusCode.Unauthorized)
                {
                    res.Headers.WwwAuthenticate.Add(
                    new AuthenticationHeaderValue("Basic", _realm));
                }
                return res;
            });
            return Task.FromResult<object>(null);
        }
    }
```

ASP.NET Web API 2.0 还提供一个名为 `HostAuthenticationFilter` 的 `IAuthenticationFilter` 具体实现，通过 Katana 身份认证管理器，使用 Katana 的身份验证中间件。

`HostAuthenticationFilter` 的 `AuthenticateAsync` 方法一开始就尝试从请求上下文中获得 Katana 身份验证管理器，如果存在身份验证管理器，筛选器就会使用这个管理器，传入已配置的认证类型，进行请求的身份验证。身份验证管理器在内部检查已注册的中间件，如

果有兼容的中间件，那么管理器会调用这个中间件，对请求进行身份验证（被动模式），或者返回中间件管道之前已经执行的身份验证结果（主动模式）。

与此类似，`HostAuthenticationFilter.ChallengeAsync` 也会尝试获得 Katana 身份验证管理器，用这个管理器添加质询信息，Katana 身份验证中间件随后会使用这些质询信息，给响应添加 `WWW-Authenticate` 标头。

15.3.11　基于令牌的身份验证

遗憾的是，我们之前介绍的基于密码的 HTTP 基础身份验证方法具有若干问题。首先，每个请求都必须发送密码，导致了如下问题。

- 客户端必须保存密码，或者每次请求时从用户获得密码（这种做法不太现实）。还要注意的是，密码必须以明文保存，或者使用一种可逆的保护方式，增加了密码泄露的危险。
- 同样的，服务器必须对每个请求都进行密码验证，会增加很多的开销。
- 验证信息通常保存在外部系统中。
- 为防御字典攻击而使用的技术，导致验证过程的计算开销高昂。
- 增加了信息偶然泄露给未授权方的可能性。

通常情况下，密码还具有很低的不确定性，很容易受到字典攻击。这意味着，任何使用密码验证的公共系统都必须采取防御字典攻击的手段。例如，限制一段时间内允许发生的错误验证次数。

密码的应用范围通常也很广，也就是说，客户端访问某个系统上任何资源，使用的都是同一个密码。大部分时候，我们应该设置一个只能访问一个资源或某些 HTTP 方法的用户身份。

而且，基于密码的机制不适合分布式应用。在分布式场景中，身份验证过程委托给外部系统，例如：组织的或社会的用户凭证提供方。最后，基于密码的身份验证不能支持委托场景，第 16 章将对此进行介绍。

进行 Web API 身份验证的一个更好的方法是使用*安全令牌*（security token）。RFC 4949 将安全令牌定义为“用于在身份验证过程中验证用户标识的（……）数据对象”。典型的 Web 应用程序中使用身份验证 cookie，就是一种基于令牌的身份验证过程：

(1) 使用基于密码的认证机制，执行初始身份验证，生成一个 cookie，返回给客户端；
(2) 客户端之后发出的每个请求，都通过这个 cookie 进行身份验证，不再需要初始的身份信息。

安全令牌是一个相当通用和抽象的概念。安全令牌可能以不同的方式进行初始化，具有不同的特征。接下来，我们将介绍一些最常见的设计变化。

首先，安全令牌可以*包含*表示的安全信息，或者只是对安全信息的一个*引用*。在二种情况中，令牌只是包含一个无法伪造的引用，指向一个安全存储条目，这个安全存储通常由令牌颁发者负责管理。这种基于引用的令牌也称为*生成物*（artifacts），基于引用的令牌具有两个优点：

- 长度更短。如果需要在 URI 中嵌入令牌，这一点非常重要；
- 更易撤销或取消。令牌颁发者只需删除这个令牌指向的存储条目即可。

但是，基于引用的令牌不是自包含的（self-contained）：如果要获取令牌所代表的安全信息，你通常需要对令牌颁发者或者外部存储进行查询。因此，当令牌颁发者和使用者是同一实体，或者令牌有长度限制时，人们才会更多使用基于引用的令牌。

令牌颁发者可以生成足够长度（例如：256 位）的随机位串，用做无法伪造的引用令牌。然后令牌颁发者可以用这个引用的散列值作为键值，存储相关的安全信息。令牌颁发者使用了密码学的散列函数，因此：

- 根据令牌内容，很容易计算出存储的键值，访问安全信息；
- 根据存储的键值，很难计算出有效的令牌，因此如果攻击者能够读取令牌存储内容，这可以提供一层额外的防御。

除了包含基于引用的令牌，令牌也可以包含安全信息，这些信息安全地进行打包，用于两个或更多方之间的通信。这些信息的打包需要使用密码机制，确保令牌具有如下特性。

- *保密性*（confidentiality）

  只有通过身份验证的接受者才能够访问所包含的信息。

- *完整性*（integrity）

  接收方应该能够检测令牌在两方之间传输时是否遭到修改。

这种令牌通常称为*断言*（assertion），具有自包含的优点：令牌使用者无需访问外部系统或存储，即可获得安全信息。这种令牌的缺点是比较长，可能会超出实际 URI 的限制，而且这种令牌的产生和使用需要使用密码机制。SAML（Security Assertion Markup Language，安全断言标记语言）断言就是一种广泛使用的自包含令牌，使用 XML 习语表示安全信息，通过 XML 签名和 XML 加密进行保护。这种安全令牌通常用于经典的联邦协议（federation protocol），例如：SAML 协议、WS-Federation 或 WS-Trust。

JWT（JSON Web Token）是一种较新的自包含令牌格式，这种格式基于 JSON 语法，目的是用于“空间受限的环境，例如 HTTP 身份验证标头和 URI 查询参数”。

下面是一个经过签名的 JWT 令牌的示例：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vaXNzdWVyLndlYmFwaWJvb
2submV0IiwiYXVkIjoiaHR0cDovL2V4YW1wbGUubmV0IiwibmJmIjoxMzc2NTcxNzAxLCJleHAiOjE
zNzY1NzIwMDEsInN1YiI6ImFsaWNlQHdlYmFwaWJvb2submV0IiwiZW1haWwiOiJhbGljZUB3ZWJhc
Glib29rLm5ldCIsIm5hbWUiOiJBbGljZSJ9.fCO6l0k_hey40kqEVuvMfiM8LeXJtsYLfNWBOvwbU-I
```

JWT 令牌有一列组成部分，各部分以 . 字符分隔。每个组成部分是一个 base64url 编码的八位字节流：前两个八位字节流是 UTF-8 编码的两个 JSON 对象，最后一个是签名方案的输出。第一个对象（在令牌第一部分编码）是 *JWT 标头*：

```
{"typ":"JWT","alg":"HS256"}
```

这个对象定义了令牌类型，以及所用的密码保护。在上面这个例子中，令牌唯一的完整性保护是通过 MAC 方案（HS256 代表 HAMC-SHA256）添加的。但是，JWT 标头也支持对令牌内容的加密。

JWT 令牌的第二部分是 JWT 声明集合（为便于阅读，示例中加入了换行符）：

```
{
  "iss":"http://issuer.webapibook.net",
  "aud":"http://example.net",
  "nbf":1376571701,
  "exp":1376572001,
  "sub":"alice@webapibook.net",
  "email":"alice@webapibook.net",
  "name":"Alice"
}
```

JWT 声明集合对象包含对*主体*（subject）的声明，这些声明由一个*颁发者*（issuer）进行断言，供*观众*（audience）使用。这个对象的每个属性都对应一个声明类型，属性值包含对应的声明值，这个值可以是任何 JSON 值（例如：字符串或数组）。JWT 规范定义了一些声明类型：

- `iss`（issure）标识令牌颁发者；
- `sub`（subject）是令牌主体的唯一标识符，令牌主体即令牌声明应用的实体；
- `aud`（audience）标识声明所允许的使用者；
- `exp`（expiration）和 `nbf`（not before）定义一个有效时间段。

示例 15-11 展示了如何使用 NuGet 软件包 `System.IdentityModel.Tokens.Jwt` 中的 `JwtSecurityTokenHandler` 类，创建和使用一个 JWT 令牌。

示例 15-11：创建和使用 JWT 令牌

```
[Fact]
public void Can_create_and_consume_jwt_tokens()
{
    const string issuer = "http://issuer.webapibook.net";
    const string audience = "the.client@apps.example.net";
```

```
    const int lifetimeInMinutes = 5;

    var tokenHandler = new JwtSecurityTokenHandler();

    var symmetricKey = GetRandomBytes(256 / 8);
    var signingCredentials = new SigningCredentials(
        new InMemorySymmetricSecurityKey(symmetricKey),
        "http://www.w3.org/2001/04/xmldsig-more#hmac-sha256",
        "http://www.w3.org/2001/04/xmlenc#sha256");

    var now = DateTime.UtcNow;

    var claims = new[]
    {
        new Claim("sub", "alice@webapibook.net"),
        new Claim("email", "alice@webapibook.net"),
        new Claim("name", "Alice"),
    };

    var token = new JwtSecurityToken(issuer, audience, claims,
        new Lifetime(now, now.AddMinutes(lifetimeInMinutes)), signingCredentials);

    var tokenString = tokenHandler.WriteToken(token);

    var parts = tokenString.Split('.');
    Assert.Equal(3, parts.Length);

    var validationParameters = new TokenValidationParameters()
    {
        AllowedAudience = audience,
        SigningToken = new BinarySecretSecurityToken(symmetricKey),
        ValidIssuer = issuer,
    };

    tokenHandler.NameClaimType = ClaimTypes.NameIdentifier;
    var principal = tokenHandler.ValidateToken(tokenString, validationParameters);

    var identity = principal.Identities.First();

    Assert.Equal("alice@webapibook.net", identity.Name);
    Assert.Equal("alice@webapibook.net",
        identity.Claims.First(c => c.Type == ClaimTypes.NameIdentifier).Value);
    Assert.Equal("alice@webapibook.net",
        identity.Claims.First(c => c.Type == ClaimTypes.Email).Value);
    Assert.Equal("Alice", identity.Claims.First(c => c.Type == "name").Value);
    Assert.Equal(issuer, identity.Claims.First().Issuer);
}
```

在验证方，TokenValidationParameters 定义了令牌使用参数，例如：允许的目的地（观众）和颁发者。这些参数也定义了签名验证密钥。这个示例使用了对称签名方案，因此令牌生成方和使用方必须使用同样的密钥。如果验证通过，验证方还会产生一个包含令牌声明的用户对象。

区分令牌的另一个特征是其绑定到信息的方式，最常见的两种方式是*持有者*（bearer）和*密钥所有者*（holder-of-key）。

RFC 6750 对持有者令牌的定义如下：

> *一种安全令牌，持有这种令牌的任何一方（一个“持有者”）具有同等的使用权。使用持有者令牌不要求持有者证明其拥有密钥（所有权证明）。*

令牌和消息之间无需额外的绑定，持有者令牌即可加入一个消息中。这意味着能够访问这个纯文本信息的任何一方都可以获得其中包含令牌，无需任何额外知识便可将其用于另一个消息。在这方面，持有者令牌与不记名支票（bearer check）很相似，二者的使用都不需要对使用者进行任何附加的身份证明。

持有者令牌很容易使用，但是，其安全性完全依赖以下条件：

- 包含令牌的消息的保密性；
- 确保令牌不会发送给错误的接收者。

令牌的另一种绑定方式是使用*密钥持有者*（holder-of-key）方法。使用这种方法，对每个通过身份验证的消息，客户端都必须提供绑定到令牌的密钥信息。客户端通常的实现方法是，选择消息的一部分内容，使用密钥计算出一个对称签名（消息认证码），加入消息中。

和使用基础认证方案一样，使用持有者令牌的客户端和服务器也共享一个秘密。但是，这个秘密实际上是用于对消息进行签名和验证的一个密钥，这个密钥*并不发送*。

- 客户端使用共享密钥，对请求消息中精心选择部分进行签名，并将这个签名附加到消息上，然后将消息和令牌（类似用户名）一起发送至服务器。
- 服务器使用这个令牌获取客户端的共享秘密，用于验证消息签名。

这种方案基于一个假设，即：对于一个密码签名机制，只有知道共享密钥的一方能够生成有效的签名。因此，客户端不需要提供这个密钥，就能证明自己拥有密钥。

如果一个恶意的第三方捕获了请求消息，那么只能得到消息签名的值，而无法获得共享密钥，因此无法给新消息提供身份验证信息。但是，第三方捕获的这个消息签名是有效的，因此可以将其进行转发。为了对此进行防御，你可以组合使用时间戳和 Nonces。

*时间戳*是一个时间值，表明源消息产生的时间。时间戳可以添加到待发送的信息，也受到签名的保护。在服务器端，只有当消息的时间戳位于接受窗口内（例如：当前时间加上或减去 5 分钟）时，服务器才接受这个消息。服务器使用这个时间窗口，以允许消息传送的延迟和时钟误差。

Nonce（“number used only once”的缩写）通常是一个只使用一次的随机数。Nonce 与时间戳一起使用，可以避免在服务器接受窗口内进行消息转发，服务器保存所有接收消息的 Nonce，如果一个消息的 Nonce 已经存在就会遭到拒绝。Nonce 与时间戳一起使用时，只需要保存接受窗口的时间长度即可。

签名机制可以分为两种。非对称机制中，签名的生成算法和验证算法使用不同的密钥——签名生成使用私有密钥，签名验证使用公用密钥。在对称机制中，签名生成和验证都使用同样的密钥。消息认证码通常使用对称机制。这种对称性意味着，任何能够验证签名的一方也能生成签名，这与传统签名不同，使得我们无法确定签名者的身份。但是，对称机制具有很好的性能，因此通常在不需要非对称机制提供的额外功能时使用。如果密钥持有证据是基于消息认证码的，那么这些令牌被称为 MAC 令牌。对称机制通常也具有确定性，同一个消息使用同一个密钥总是会产生通常的签名值。因此，我们可以计算签名值，与接收到的消息的签名值进行比较，以此进行签名验证。

构建 MAC 算法的一个常用方法是 RFC 2014 中定义的 HMAC（hash-based message authentication code，基于散列的消息认证码）。HMAC 内部使用了一个密码学散列函数。例如，Amazon S3 使用了 HMAC 与 SHA-1 函数的组合，称为 HMAC-SHA1。Windows Azure Blob Service 也使用了 HMAC 算法，但是使用了更新版本的 SHA-256 散列函数（HMAC-SHA256）。

MAC 令牌技术用在若干身份验证方案中，其中有：

- Amazon Simple Storage Services（S3）（参见 http://s3.amazonaws.com/doc/s3-developer-guide/RESTAuthentication.html）；
- Windows Azure Storage Services（参见 http://msdn.microsoft.com/library/azure/dd179428.aspx）；
- OAuth 1.0 协议（参见 http://tools.ietf.org/html/rfc5849）；
- 由 Eran Hammer 提出的 Hawk HTTP 身份验证方案（参见 https://github.com/hueniverse/hawk）。

这四种方案都使用生成物令牌，也就是说令牌只是唯一标识符，服务器用其获取客户身份声明和令牌密钥。

图 15-10 展示了基于签名的身份验证过程。发送方从消息中抽取一个*消息代表*（稍后进行定义），使用共享密钥对其进行签名，并将生成签名值插入待发送的请求消息。接收方抽取消息代表，对其进行签名，将签名值进行比较，从而验证这个签名。

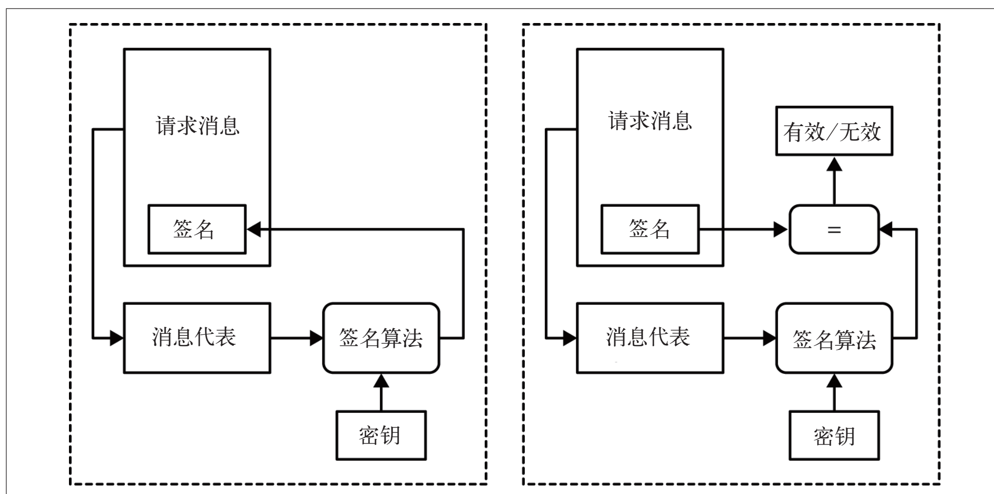


图 15-10：消息签名计算

HTTP 中间件可以修改请求消息的部分内容（例如：去除代理标头），因此我们不能对整个消息进行签名，而是对消息代表进行签名。消息代表（message representative）是从消息构建得到的一个字符串，具有如下特性：

- 不受 HTTP 中间件对消息的典型修改的影响；
- 捕获了消息的所有重要部分。两个语义不同的消息不应该有相同的消息代表。

15.3.12　Hawk身份验证方案

为了使这些抽象的概念具体化，这一节我们要简要介绍 Hawk 身份认证方案。在 Hawk 身份认证方案中，客户端使用换行符连接如下元素组成消息代表：

- 字符串常量 `"hawk.1.header"`；
- 一个时间戳字符串，代表从 GMT 时间 1970 年 1 月 1 日 0 时 0 分 0 秒到现在的秒数；
- 一个 Nonce；
- 请求 HTTP 方法；
- 请求 URI 路径和查询；
- 请求 URI 主机名（不包括端口）；
- 请求 URI 端口；
- 可选的有效载荷散列值或空字符串；
- 可选的应用相关的扩展数据或空字符串。

消息代表在构建之后，使用 UTF8 编码转换成一个字节序列，然后提供给配置令牌密钥的 MAC 方案。与 Amazon 和 Azure 身份验证方案不同，Hawk 方案支持多个 MAC 算法

（目前支持 HMAC-SHA1 和 HMAC-SHA256）。MAC 方案的输出（一个字节序列）后通过 Base64 编码算法，转换回字符串。

请求消息的 `Authorization` 标头使用 `Hawk` 方案字符串，其后跟随一组键 / 值对。

- `id`
  令牌 ID。

- `ts`
  所用的时间戳。

- `nonce`
  所用的 Nonce。

- `mac`
  MAC 输出的 Base64 编码字符串。

- `hash`
  （可选）有效载荷代表的散列值。

- `ext`
  （可选）可选的扩展数据。

其中时间戳、Nonce 和扩展数据必须明确添加到消息的 `Authorization` 标头，使服务器可以重新构建消息代表。服务器使用 `id` 字段指明的密钥，从这个消息代表重新计算出 MAC 输出，然后将其与接收到的 `mac` 字段进行比较。如果比较的 MAC 值不同，那么说明消息遭到篡改，应该拒绝接受。然而，只比较 MAC 值是不够的，攻击者可能会转发过去的有效信息。为了防御这种攻击，服务器应当采取以下操作。

- 检查接收到的 Nonce，确保其未曾在以前的消息中使用过。
- 检查接收到的时间戳，确保其位于一个接受时间窗口内。这个时间窗口的默认值大约是一分钟。

服务器还应当存储接收到的 Nonce，这些值应该至少保存接受时间窗口的长度。

Hawk 身份验证方案允许消息代表包含有效载荷代表的散列值，以此对请求消息的有效载荷进行保护。有效载荷代表是以换行符连接如下元素：

- 字符串常量 `"hawk.1.payload"`；
- 请求内容类型（例如：`application/xml`），其中不包含参数；
- 在进行任何内容或传输编码之前的请求有效载荷。

使用有效载荷保护时，有效载荷代表字符串的散列值也包含在 `Authorization` 标头（`hash`

字段）中，使服务器可以在计算有效载荷散列值之前验证消息代表。

要了解 Hawk 身份验证方案的更多细节，你可以访问 https://github.com/hueniverse/hawk，这里既有 Hawk 的非正式描述，也有一个基于 Node.JS 的具体实现。本书的 GitHub 存储也包含一个 C# 的 Hawk 方案实现，称为 HawkNet。

下一章主要介绍 OAuth 2.0 框架，提供了更多具体的示例，演示基于令牌的身份验证，也就是获取和使用令牌的协议。

15.4　授权

我们已经了解到，身份认证处理的问题是收集和验证关于主体的信息，也就是身份声明。而授权处理的则是一个附加问题，即控制这些*主体*（subject）对受保护*资源*（resource）可以实施的*操作*（action）。图 15-11 展示了授权问题的核心概念：主体、操作和资源。

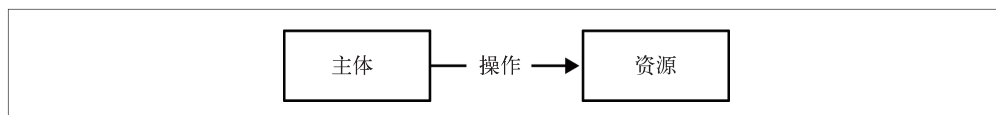


图 15-11：基础授权模式：主体、操作和资源

主体、操作和资源的具体组成很大程度上依赖具体的上下文。例如，.NET 框架提供一个代码访问授权模型，其中资源是类方法，操作对应方法调用，主体则是与运行线程相关的用户身份。而在 Web API 中，这些概念映射则颇为直接：

- 受保护的资源对应于请求消息要访问的 HTTP 资源；
- 操作是 HTTP 方法（例如：`GET` 或 `DELETE`）；
- 最后，主体对应于执行 HTTP 请求的 HTTP 客户端。

人们经常将授权问题分为几个部分：策略、决定和执行。*授权策略*（authorization policy）是定义允许情况的规范。下面的声明就是用自然语言表达的一个授权策略的示例：

- “匿名主体不能执行不安全的 HTTP 方法”；
- “问题只能由创建者或项目经理关闭”；
- “工作单的标题只能由创建者修改”。

授权决定（authorizatino decision）是一个过程，衡量由（主体，操作，资源）三元组定义的一个访问是否符合规定的授权策略。最后，*授权执行*（authorization enforcement）是确保系统只执行得到允许的访问的机制。授权执行通常与拦截访问的运行时机制（例如：Web API 筛选器）关系紧密，而授权决定则依赖授权策略。图 15-12 展示了这些概念及其关系。

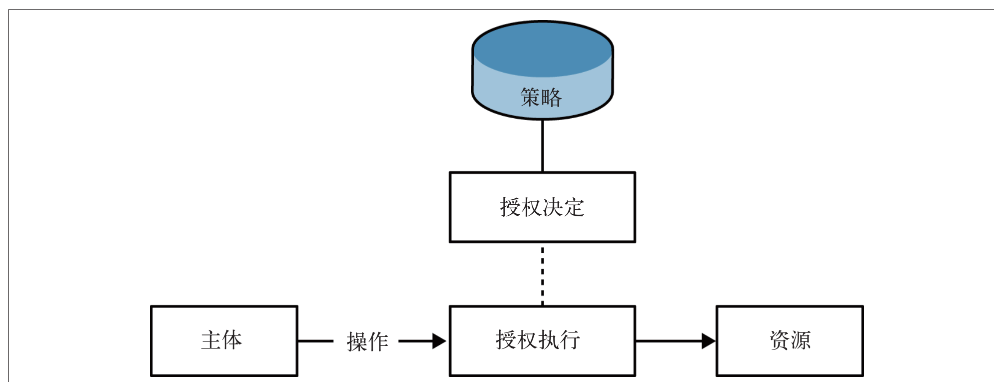


图 15-12：授权执行、授权决定和授权策略

在分布式系统中，授权决定和授权执行可能在不同的节点执行，或在多个层次上执行。例如：使用 OAuth 2.0 框架（详见下一章）时，你可以在一个外部授权服务器（authorization server）上进行授权决定，让资源服务器（resource server）负责确保只有得到允许的访问能够执行。

授权也可以在多个层次上执行，即靠近外部世界的连接点（Web API），或靠近域对象和方法。这些层次经常是互相补充的。如果在 Web API 层执行授权，我们可以尽早终止未授权请求的处理，但策略决定点可能无法访问所有需要的领域信息，不能做出全面的决定。另一方面，在领域层执行授权可以访问更丰富的信息，但是也意味着未授权请求会耗费较多的计算资源。本章关注的重点是 Web API 层的授权。

15.4.1 授权执行

ASP.NET Web APi 提供多种拦截 HTTP 请求的方法，其中一种是授权筛选器。授权筛选器位于控制器层，位于认证筛选器之后，模型绑定和操作筛选器之前。看名字就知道，授权筛选器可以用于执行授权执行。ASP.NET Web API 提供一个具体的授权筛选器 `AuthorizeAttribute`，可以用于标记控制器类或操作。`AuthorizeAttribute` 类有两个定义授权策略的属性。

- `User` 是逗号分隔的用户名列表。如果 `User` 不为空，那么只有当用户名在列表中时，才允许访问。
- `Roles` 是逗号分隔的角色列表。如果 `Roles` 不为空，那么只有当用户所属某个角色在列表中时，才允许访问。

示例 15-12 展示了如何使用 `AuthorizeAttribute`。

- `ResourceController` 类的 `AuthorizeAttribute` 标记，要求所有请求都进行认证。`GET` 请求是唯一的例外，因为 `Get` 操作标记为 `AllowAnonymous`。

- Delete 操作的 Authorize(Roles = "ProjectManager") 标记，要求所有的 DELETE 请求都必须由 ProjectManager 角色的主体执行。

示例 15-12：使用 AuthorizeAttribute

```
[Authorize]
public class ResourceController : ApiController
{
    [AllowAnonymous]
    public HttpResponseMessage Get()
    {
        return new HttpResponseMessage
        {
            Content = new StringContent("resource representation")
        };
    }
    public HttpResponseMessage Post()
    {
        return new HttpResponseMessage()
        {
            Content = new StringContent("result representation")
        };
    }
    [Authorize(Roles = "ProjectManager")]
    public HttpResponseMessage Delete(string id)
    {
        return new HttpResponseMessage(HttpStatusCode.NoContent);
    }
}
```

可是，直接在 AuthorizeAttribute 中定义授权策略的做法，使授权策略与资源控制器绑定在了一起，导致任何的策略修改（例如："QA 工程师也可以删除工作单"）都需要修改和重新编译代码。

更为合理的做法应该是让授权属性将授权决定委托给外部组件，这个外部组件就可以独立地演化和部署。一种实现方法是使用*声明授权管理器*（claims authorization manager），.NET 4.5 通过基类 ClaimsAuthorizationManager（参见示例 15-13）提供声明授权管理器的概念。授权声明管理器的主要作用是，通过 CheckAccess 方法执行授权决定。在默认情况下，CheckAccess 方法返回 true，但是派生类可以重写这个方法，实现定制的授权策略。

示例 15-13：基类 ClaimsAuthorizationManager

```
public class ClaimsAuthorizationManager : ICustomIdentityConfiguration
{
        public virtual bool CheckAccess(AuthorizationContext context)
        {
            return true;
        }
        ...
}
```

CheckAccess 方法的参数是一个 AuthorizationContext 类（参见示例 15-14），这个类通过（主体，操作，资源）三元组表示一个访问，其中资源由 ClaimsPrincipal 表示。有趣的是，操作和资源都由声明表示。还要注意的是，AuthorizationContext 类与授权执行机制无关，也就是说，AuthorizationContext 类独立于 Web API 或其他类似技术。

示例 15-14：描述访问的 AuthorizationContext 类

```
public class AuthorizationContext
{
    public ClaimsPrincipal Principal
    {
            get {...}
    }

    public Collection<Claim> Action
    {
            get {...}
    }

    public Collection<Claim> Resource
    {
            get {...}
    }

    public AuthorizationContext(ClaimsPrincipal principal,
                                Collection<Claim> resource,
                                Collection<Claim> action)
    {...}
}
```

程序库 Thinktecture.IdentityModel.45（https://github.com/thinktecture/Thinktecture.IdentityModel）提供了一个 ClaimsAuthorizAttribute，使用策略为：当 Web API 运行时调用这个属性，检查请求是否得到允许时，这个属性将此决定委托给已注册的单例 ClaimsAuthorizationManager。ClaimsAuthorizAttribute 使用一个授权上下文参数，其中包含了请求的声明用户对象、操作名和控制器名。示例 15-15 实现了一个定制的授权管理器，实现示例 15-12 中定义的授权策略。这两个示例的主要区别在于，示例 15-15 将授权策略外部化，在一个单独的组件中实现，这个组件可以独立演化和编译。

示例 15-15：一个定制的 ClaimsAuthorizationManager 类

```
public class CustomPolicyClaimsAuthorizationManager : ClaimsAuthorizationManager
{
    public override bool CheckAccess(AuthorizationContext context)
    {
        var subject = context.Principal;
        var method = context.Action
            .First(c => c.Type == ClaimsAuthorization.ActionType).Value;
        var controller = context.Resource
            .First(c => c.Type == ClaimsAuthorization.ResourceType).Value;

        if (controller == "ClaimsResource")
```

```
            {
                if (method.Equals("GET", StringComparison.OrdinalIgnoreCase))
                    return true;

                if (method.Equals("DELETE", StringComparison.OrdinalIgnoreCase)
                        && !subject.IsInRole("ProjectManager"))
                    return false;

                return subject.Identity.IsAuthenticated;
            }
            return false;
        }
    }
```

下一章将接着讨论授权，介绍 OAuth 2.0 框架。在那之前，我们先来看另一种授权：控制浏览器对跨域资源的访问。

15.4.2 跨域资源共享

用户代理，如浏览器，对来自多个资源的内容（例如：HTML 文档和脚本程序）进行组合及处理。通常情况下，这些资源有着不同的信任级别，可能包含一些恶意站点，妄图破坏其他站点内容的保密性和完整性。*同源策略*（same-origin policy）是用户代理执行的一组安全策略，这种策略使用内容的源，对这些内容可以如何交互进行限制，也就是通过用户代理的内部 API 进行授权（例如：DOM 访问和网络）。

“源”（origin）的概念（参见 RFC 6454）将 URI 按照其方案、主机名和端口进行组织。简单地说，如果两个 URI 具有相同的（方案，主机名，端口）三元组，就是同源的。例如：http://example.coom/ 和 http://example.com:80/path 具有相同的源；而 http://example.com 和 http://www.example.com 具有不同的源。

`XMLHttpRequestAPI` 就属于使用同源策略的用户代理 API。当一个请求通过 `open` 方法进行初始化时，`XMLHttpRequestAPI` 对象对以下 URI 的源进行比较：

- 受到请求的 URI；
- 初始化这个 `XMLHttpRequest` 的文档的 URI。

只有当这两个 URI 具有相同的源时，这个请求才得到允许，从而禁止了*跨域*（cross-origin）请求。

因为基于 cookie 的身份信息会自动附加到每个发送的请求，因此同源策略对于浏览器上下文特别重要。例如：如果一个浏览器有访问 https://banking.example.net 的有效的身份验证 cookie，那么这些 cookie 会自动附加到任何发送到这个源的请求。如果一个来自外部源的恶意脚本得到允许，可以执行对 https://banking.example.net 的请求，那么该请求会自动通过身份验证，脚本就可以访问 https://banking.example.net 上的受保护资源。同源策略可以

防御这种类型的攻击。

但是，在有些情况下，资源允许对来自其他源内容的访问进行授权，跨域请求限制也禁止了这些合法场景。CORS（Cross-Origin Resource Sharing，跨域资源共享）是一个 W3C 的工作草案，定义了一种机制，允许待访问资源对跨域请求进行授权，从而允许跨域资源访问。CORS 基于额外的 HTTP 请求和响应标头，以及用户代理的一组处理规则，即支持 CORS 的 `XMLHttpRequest` 实现。

简单说来，CORS 规范规定，跨域请求必须由 `XMLHttpRequest` 通过两种方式执行。一种方式是使用*简单跨域请求算法*（simple cross-origin request algorithm），预先发送请求，但是只有当发起调用的脚本明确得到资源授权时，才能看见请求结果；另一种方法是使用*带有预检算法的跨域请求*（cross-origin request with preflight algorithm），预先执行一个 `OPTIONS` 请求（即预检），查询资源是否授权跨域访问。只有当得到肯定的查询结果时，才执行原本的请求。

*简单*算法是只针对于跨域请求的一项优化，即使没有 CORS，脚本也可能已经发起了跨域请求。例如，一个脚本程序可以使用动态生成并提交的 HTML 表单，发起跨域 `GET` 或 `POST` 请求。这意味着，资源必须做好准备，正确处理基于跨域请求的攻击，这种攻击通常称为 CSRF（Cross-Site Request Fogery，跨站请求伪造）攻击。在这种情况下，支持 CORS 的 `XMLHttpRequest` 只需要确保，只有明确得到资源授权时，请求的响应才对脚本可见。

在满足以下条件时，跨域请求可以使用*简单*算法：

- 请求方法为 `GET`、`HEAD` 或 `POST`（所谓的*简单*方法）；
- 由脚本明确添加的请求标头都是简单标头（`Accept`、`Accept-Language`、`Content-Language` 或 `Content-Type`）；
- 内容类型为 `application/x-www-form-urlencoded`、`multipart/formdata` 或 `text/plain`。

假设我们有一个脚本，位于从 http://www.example.net 加载的一个文档中，想要使用 `XMLHttpRequest` API，访问位于 https://api.example.net 的一个资源（主机名和方案不同，因此脚本和资源的源不同）。如果满足简单算法的所有限制条件，支持 CORS 的用户代理就会预先发送这个请求，在请求中加入一个 `Origin` 标头，标头值为文档的源（http://www.example.net）：

```
GET https://api.example.net/api/resource HTTP/1.1
Host: api.example.net
Origin: http://www.example.net
Referer: http://www.example.net/
```

这时由资源负责判断，源 http://www.example.net 是否有权访问资源。如果有权访问，那么返回的响应必须包含一个 `Access-Control-Allow-Origin`，表示赋予这个源访问权限：

```
HTTP/1.1 200 OK
Content-Length: 23
Content-Type: text/plain; charset=utf-8
Access-Control-Allow-Origin: http://www.example.net

resource representation
```

当用户代理接收这个响应，并确认响应中包含了对请求者源的 Access-Control-Allow-Origin 后，就将这个响应交给发起调用的脚本。如果响应中没有 Access-Control-Allow-Origin 或者不包含调用者的源，那么用户代理就会向发起调用的脚本传回一个网络错误。请注意，如果资源不支持 CORS，那么响应中不会包含 Access-Control-Allow-Origin，用户代理回将其解释为拒绝访问。

如果简单算法的条件中有任何一个不满足（例如：请求使用 PUT 或 DELETE 方法），那么 CORS 规范使用一个预检请求：用户代理首先对资源执行一个 OPTIONS 请求，检查该资源是否支持跨域请求。这个请求包含一个值为调用者的源的 Origin 标头，以及值为所请求的 HTTP 方法的 Access-Control-Request-Method：

```
OPTIONS https://api.example.net/api/resource HTTP/1.1
Host: api.example.net
Access-Control-Request-Method: PUT
Origin: http://www.example.net
Access-Control-Request-Headers: origin
Referer: http://www.example.net/
```

支持 CORS 的资源会使用这个源以及方法信息，判断是否允许跨域访问。如果允许访问，该资源会生成一个包含 Access-Control-Allow-Origin 和 Access-Control-Allow-Methods 标头的响应消息。

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.example.net
Access-Control-Allow-Methods: PUT
Content-Length: 0
```

只有接收到这个表明允许访问的响应后，用户代理才会执行原本的请求。

```
PUT https://api.example.net/api/resource HTTP/1.1
Host: api.example.net
Connection: keep-alive
Origin: http://www.example.net
Referer: http://www.example.net/
```

同样，只有当响应中包含值为脚本源的 Access-Control-Allow-Origin 标头时，用户代理才会将响应消息交给发起调用的脚本。

```
HTTP/1.1 204 No Content
Access-Control-Allow-Origin: http://www.example.net
```

为了进行优化，资源也可以在预检响应中包含一个 `Access-Control-Max-Age`，允许用户代理在该标头定义的时间内，将这个响应保存在一个*预检结果缓存*（preflight result cache）中。这个功能可以减少需要执行的预检请求数量。

15.4.3 ASP.NET Web API的CORS支持

ASP.NET Web API 2.0 增加了对 CORS 规范的支持，提供定义跨域策略和执行机制的方法。你可以调用 `HttpConfiguration` 对象的 `EnableCors` 扩展方法，激活全局的跨域支持。

```
config.EnableCors();
```

然后，你可以使用 `EnableCorsAttribute`，明确标记应该支持 CORS 的控制器或操作。

```
[EnableCors(...)]
public class ResourceController : ApiController
{
    ...
}
```

你也可以向 `EnableCors` 方法传入一个 `EnableCorsAttribute` 实例，定义全局的跨域支持。

```
config.EnableCors(new EnableCorsAttribute(...));
```

`EnableCorsAttribute` 不仅激活了跨域支持，而且定义了允许的跨域*策略*（例如：允许的源或者请求方法集合）。为此，`EnableCorsAttribute` 的构造函数参数有：允许的源、允许的请求方法，以及允许和提供的标头。`EnableCorsAttribute` 还可以定义预检过期时间。

```
[EnableCors(origins:"https://localhost", headers:"*", methods:"GET",
    PreflightMaxAge = 60)]
public class ResourceController : ApiController
{
    ...
}
```

正如我们预期的，由标记操作的属性定义的策略，优先级高于标记控制器类的属性定义的策略。传给 `config.EnableCors` 的属性定义的是默认策略，当请求的控制器和操作都没有相关策略时，系统就会使用默认策略。

除了这个简单属性模型，Web API 内部还提供一个可扩展基础结构（参见图 15-13），可以用别的方式定义跨域策略。

跨域策略由 `CorsMessageHandler` 执行。`CorsMessageHandler` 由 `EnableCors` 方法插入请求管道中，当一个 HTTP 请求到达时，这个处理程序检查 `Origin` 标头，验证是否支持 CORS。如果支持 CORS，那么这个处理程序会构建一个 `CorsRequestContext`，其中包含 CORS 相关的请求信息。

```
public class CorsRequestContext
{
    public Uri RequestUri { get; set; }
    public string HttpMethod { get; set; }
    public string Origin { get; set; }
    public string Host { get; set; }
    public string AccessControlRequestMethod { get; set; }
    public bool IsPreflight {get;}
    // ...
}
```

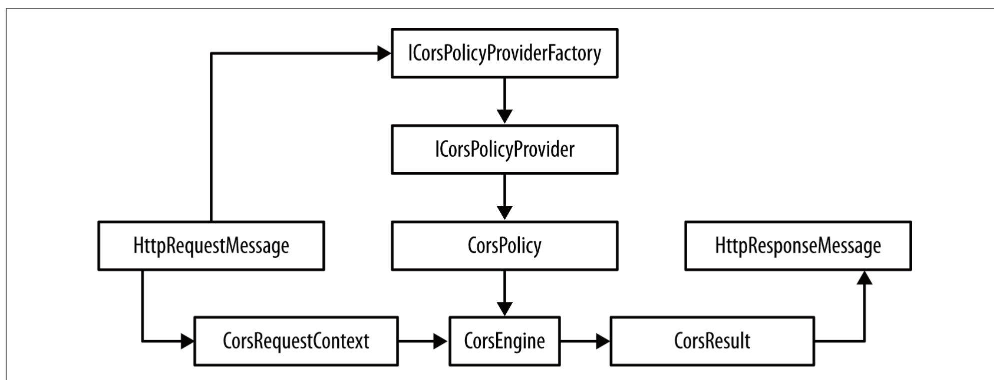


图 15-13：CORS 运行时

随后，这个处理程序使用配置中注册的 ICorsPolicyProviderFactory，尝试找到该请求的策略提供程序。

```
public interface ICorsPolicyProviderFactory
{
    ICorsPolicyProvider GetCorsPolicyProvider(HttpRequestMessage request);
}

public interface ICorsPolicyProvider
{
    Task<CorsPolicy> GetCorsPolicyAsync(HttpRequestMessage request,
    CancellationToken cancellationToken);
}
```

这两个接口是必须的，因为工厂类是静态定义的，而每个请求的策略提供程序可能不同。

跨域策略是下面的类的一个实例：

```
public class CorsPolicy
{
    public bool AllowAnyHeader { get; set; }
    public bool AllowAnyMethod { get; set; }
    public bool AllowAnyOrigin { get; set; }
    public IList<string> ExposedHeaders { get; private set; }
    public IList<string> Headers { get; private set; }
```

```
        public IList<string> Methods { get; private set; }
        public IList<string> Origins { get; private set; }
        public long? PreflightMaxAge { get; set; }
        public bool SupportsCredentials { get; set; }
    }
```

CorsMessageHandler 使用这个 CorsPolicy 实例，将请求的 CorsRequestContext 转换为一个 CorsResult，然后应用于 HTTP 响应。因此，你可以定义一个新的 ICorsPolicyProviderFactory，在配置中进行注册（扩展方法 SetCorsPolicyProviderFactory 可以完成这项任务），完全改变 CORS 策略的定义方式。

默认情况下，AttributeBasedPolicyProviderFactory 类实现 ICorsPolicyProviderFactory 接口，检查控制器和操作描述符是否带有 EnableCorsAttribute 属性。具体实现根据 CORS 请求类型而不同。对于非预检请求，CorsMessageHandler 处理程序首先将请求转发到其内部处理程序。当响应返回时，处理程序使用选中的操作描述符（保存在请求属性中），判断是否有与操作或控制器关联的 EnableCorsAttribute。如果有，处理程序使用这个 EnableCorsAttribute 获得策略——EnableCorsAttribute 实现了 ICorsPolicyProvider——并将生成的 CorsResult 应用于返回的响应。这一操作给返回的响应消息添加了附带的 CORS 标头。

但是，对于预检请求，操作略有不同。请记住，预检请求使用 OptionsHTTP 方法，用于探测服务器似乎否对给定的请求 URI 和方法对（方法在 Access-Control-Request-Method 标头中）提供 CORS 支持。也就是说，处理预检请求时服务器不应该执行任何操作。因此，当服务器收到一个预检请求时，CorsMessageHandler 会将 OPTIONS 方法替换为 Access-Control-Request-Method 中指定的方法，然后使用 Web API 解析服务，找到映射到该请求的控制器和操作。但是，这个控制和操作不会得到调用，而只是用于找到和获取 CORS 策略（由 `EnableCorsAttribute 提供）。最后，处理程序创建和返回预检响应，提前结束任何上层栈中的处理。简言之，这些预检请求永远不会到达控制器层。

15.5　小结

本章介绍了在设计、实现和使用 Web API 时必须解决的一些安全问题。我们主要关注了与 Web API 相关的安全概念和技术：传输安全、身份认证和授权。下一章将继续进行安全主题，介绍 OAuth 2.0 框架。但是，虽然其他一些安全主题没有包括在这本书中，我们也不应该将其忽略。与 Web 应用程序类似，在大部分时候，Web API 是外部因特网与关键业务内部系统之间的连接点。因此，安全编码实践，例如：输入验证、适当的输出编码和各种注入攻击的防御（如 SQL 注入），仍然是至关重要的。

第16章

OAuth 2.0授权框架

被授予的权力不能再次被授出。

RFC 6749 定义的 OAuth 2.0 授权框架，是 OAuth 1.0 协议的演化版本。在这本书编写时，已经有多个流行的 Web API 使用了 OAuth 2.0 授权框架，例如：Google API、Facebook 和 GitHub。OAuth 2.0 授权框架主要应用于委托的受限授权。请看图 16-1 中的虚构场景。

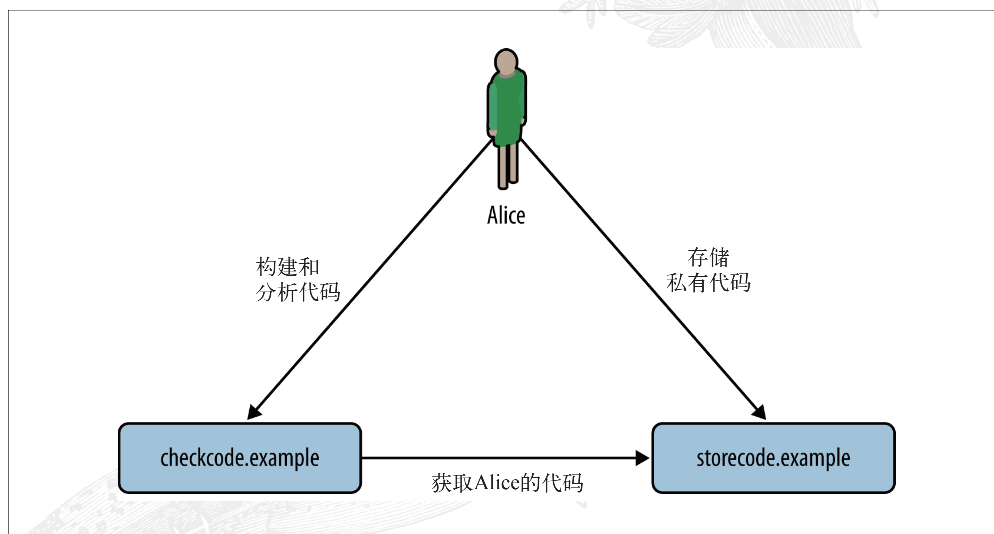


图 16-1：委托授权场景

在这张图中，你可以看到：

- storecode.example 是一个存储和管理代码的网站，并提供相关的 Web API；
- checkcode.example 是一个构建和分析代码的服务，并提供一些功能，例如持续集成、编码规则检查、错误估计以及测试覆盖；
- Alice 使用 storedcode.example 网站，存储和管理自己的私有代码。

Alice 希望使用 checkcode.example 服务，分析自己存储在 storedcode.example 的代码。storedcode.example 提供一个 API，使得这个使用场景成为可能，但是还存在一个问题：Alice 怎样允许 checkcode.example 访问她的一些私有代码存储？

要解决这个问题，Alice 可以向 checkcode.example 提供自己访问 storecode.example 所使用的身份信息（例如：用户名和密码），checkcode.example 就可以访问 Alice 的私有代码。但是，这种做法存在如下几个缺点。

- 持有这些身份信息，checkcode.example 可以执行任何 Alice 有权进行的操作，包括访问 Alice 的所有代码，还可以进行修改。换句话说，这种做法赋予了 checkcode.example 对 storecode.example *不受限的权限*（unconstrained authorization）。
- 如果 checkcode.example 受到攻击，可能会泄露 Alice 的密码，攻击者就可以对 Alice 的资源拥有完全控制。
- 如果 Alice 要撤销 checkcode.example 的权限，只能修改自己的身份信息。而这么做，其他有权访问 Alice 代码的应用程序（例如：提供托管服务的 www.hostcode.example）的权限也会撤销。

一个更好的解决办法是，Alice 向 checkcode.example 赋予一个*受限授权*（constrained authorization），只允许 checkcode.example 在一个限定的时间段内执行一部分操作（例如，读取一个代码库的主分支）。Alice 还应当能够随时撤销这个授权，而不会影响访问她的资源的其他服务。

> **非虚构场景**
>
> 在我们编写这本书时，AppHarbor PaaS（Platform as a Service，http://support.appharbor.com/kb/3rd-party-integrations/integrating-with-github） 和 Travis CI（http://docs.travis-ci.com/user/getting-started/）持续继承服务都使用 OAuth 2.0 和委托授权，与 GitHub 存储库进行集成。

之前这个示例说明了，使用简单客户端 – 服务器模型表达 Web API 授权需求的不足之处。也就是说，因为 Web API 是应用程序使用的接口，所以授权模型的一个重要功能是，区分客户端应用程序和人类用户。为此，OAuth 2.0 框架引入了一个具有四个角色的模型。

- Alice 扮演的角色是*资源所有者*——拥有受保护资源，或者能够授予访问受保护资源权限的实体。在本章后面的部分，我们将把资源所有者简单称作*用户*。

- storecode.example 扮演的角色是资源服务器——提供访问受保护资源的接口的实体。
- checkcode.example 扮演角色的是客户端——代表资源所有者访问受保护资源的应用程序。
- 授权服务器是第四个角色，负责管理授权和颁发权限。通常情况下，这个角色由资源服务器兼任。但是，OAuth 2.0 框架允许单独部署这些角色。

这个模型的一个主要特点是，在这一场景中用户和客户端不同义。用户和客户端是单独的实体，各自有完整定义的身份，大多数时候分属不同的安全领域。例如，对受保护资源的访问经常涉及与资源所有者（用户）和代表所有者执行访问的应用程序（客户端）。OAuth 2.0 模型的这一特点与之前介绍的简单的客户端 – 服务器场景完全不同。但是，虽然重点是委托授权和用户 – 客户端 – 服务器模型，OAuth 2.0 框架也支持较为简单的场景，即客户端自发进行资源访问，没有用户的介入。

Thinktecture 授权服务器

Thinktecture 授权服务器（参见 https://github.com/thinktecture/Thinktecture.Authorization Server）是一个开源的 OAuth 2.0 授权服务器，基于 .NET 平台，以 C# 编写。Thinktecture 授权服务器独立于具体的资源服务器或用户凭证提供方，适用于更广泛的应用场景。作为开源软件，Thinktecture 授权服务器是学习 OAuth 2.0 设计和实现细节的极佳学习材料。因此，在本章中我们将以 Thinktecture 授权服务器为例进行讲解。为简单起见，我们将 Thinktecture 授权服务器简称为 T.AS。

16.1 客户端应用程序

OAuth 2.0 框架设计为支持不同类型的客户端应用程序，例如：

- 经典的服务器端 Web 应用程序；
- 本地应用程序，特别是移动应用；
- 基于 JavaScript 的客户端 Web 应用程序，如 SPA（Single-Page Application，单页应用程序）。

如此多的客户端类型带来了不同的挑战，特别是对于身份验证信息长期存储的处理。为了加入 OAuth 2.0 部署，一个客户端必须预先在授权服务器注册。在典型的 OAuth 2.0 场景中，客户端所有者会提供一组信息，例如：

- 供人类读取的描述信息，例如应用程序名称、商标、主页或版本信息；
- 协议中用到的技术信息，例如重定向 URI 或所需的授权范围。

另一方面，授权服务器给客户端分配一个 `client_id` 字符串作为唯一标识。有些客户端可能还会收到一个 `client_secret` 字符串，可以在一些协议步骤中在认证服务器上进行身份验证。在这种情况下，OAuth 2.0 将客户端分为如下两类。

- 保密客户端可以安全存储 `client_secret`，将其用在协议步骤中。这种客户端的典型例子是传统的服务器端 Web 应用程序，客户端的凭证就存储在服务器端。
- 公共客户端无法安全保存用户凭证。这种客户端没有 `client_secret`，但是仍然有分配的 `client_id`。公共客户端的典型例子是客户端的 JavaScript 应用程序，客户端的 JavaScript 应用程序完全在用户的浏览器中运行，因此不能安全保存长期凭证。

根据客户端所有者提供的输入信息，在注册时客户端通常归类为保密客户端和公共客户端。

在我们编写这本书时，通常是由客户端所有者通过 Web 表单进行客户端注册，表单如图 16-2 所示。

图 16-2：GitHub 的客户端注册表单（2013）

但是，OAuth IETF 工作组（参见 https://datatracker.ietf.org/doc/draft-ietf-oauth-dyn-reg/）也在制定一个规范，定义如何使用 Web API 动态注册客户端。

授权服务器也可以将授权策略与已注册客户端进行关联，限制客户端的权限。例如，在图 16-3 的 T.AS 客户端模型中，我们可以看到，一个客户端与下列概念相关联：

- 客户端可以参与的 OAuth 2.0 流；
- 客户端可以得到委托的一组授权——即范围（稍后进行介绍）。

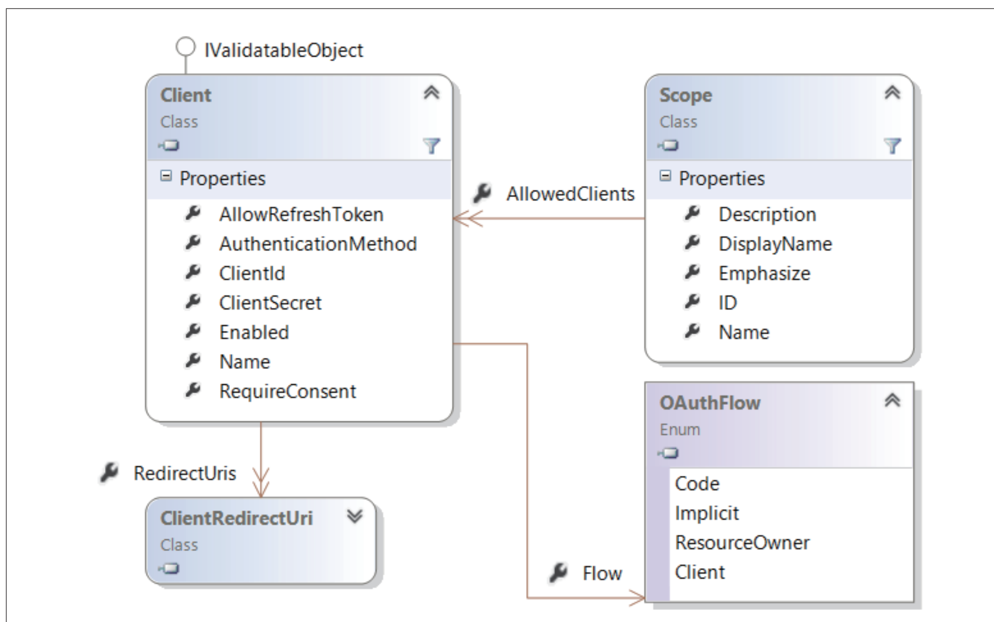


图 16-3：T.AS 的客户端类模型

16.2　访问受保护资源

在 OAuth 2.0 框架中，客户端访问受保护资源时，必须提供一个*访问令牌*（access token）（参见图 16-4）。在我们编写这本书时，OAuth 2.0 框架只定义了持有者令牌的用法[1]，即：将访问令牌简单添加到请求消息中，无需进行更多的绑定。我们前面介绍过，持有者令牌的使用较为简单，但是具有若干安全缺点。特别是，为使用传输安全，客户端必须确保令牌只发送到关联的资源服务器时，不应该使用持有者令牌。因为持有者令牌的这些局限，OAuth IETF 工作组也在制定基于 MAC 访问令牌的规范（参见 http://tools.ietf.org/wg/oauth/draft-ietf-oauth-v2-http-mac/）：使用访问令牌时必须计算消息认证码（message authentication code，MAC），将访问令牌与密钥的持有凭证一起使用。

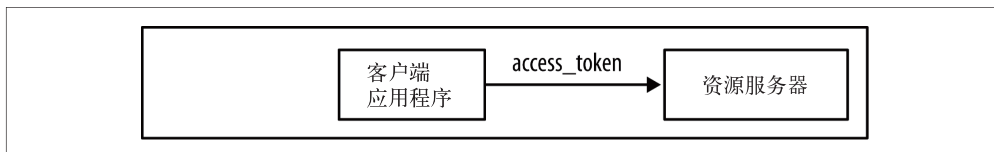


图 16-4：使用访问令牌进行资源访问

`access_token` 表示一个权限的授予，由资源服务器用于获取请求者的信息，也就是执行资

注 1：参见 RFC 6750。

源的授权策略。这个概念听起来很含糊，但是我们稍后会回到这个话题，给出访问令牌及其包含信息的具体示例。我们还将了解到，基于 ASP.NET 的资源服务器如何从请求消息中获取令牌，将其转换为用户身份和授权信息。

要将访问令牌绑定到请求消息，推荐做法是使用 `Authorization` 标头，将标头值设为 `Bearer` 方案：

```
GET https://storecode.example/resource HTTP/1.1
Authorization: Bearer the.access.token
```

这种推荐做法使用第 1 章介绍的通用的 HTTP 身份认证框架。的确，我们也可以在 `application/x-www-form-urlencoded` 正文中，或者请求 URI 的查询字符串中发送访问令牌，但这些不是推荐的做法。在请求 URI 中使用请求令牌的做法尤为不妥，因为请求 URI 通常都记录在日志中，很容易遭到泄露。

16.3 获得访问令牌

客户端应用程序可以向*令牌端点*（token end-point）请求获得令牌，令牌端点是认证服务器一部分（参见图 16-5）[1]。令牌请求包括一个*权限授予*（authorization grant），权限授予是一个抽象概念，表示认证决定所依据的信息。权限授予可以有多种实现。

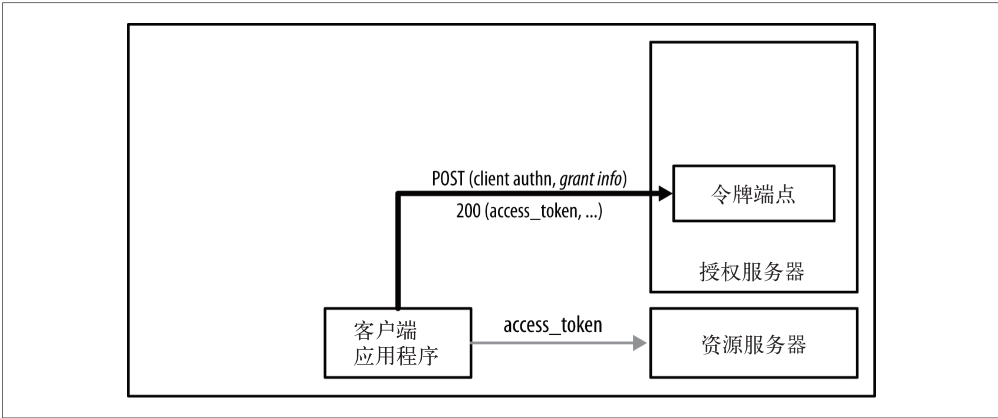


图 16-5：使用认证服务器的令牌端点获得访问令牌

在简单场景中，客户端自发访问资源服务器（没有用户介入），权限授予可以只是客户端身份信息。在 OAuth 2.0 术语中，这称为*客户端凭据授予*（client credential grant）流——授权完全基于客户端的身份信息。这个情况要求客户端是保密类型的——即：持有分配的 `client_secret`。

如果有用户介入，权限授予可以基于用户的密码凭据，这一凭据由用户提供给客户端，如

注 1：隐式流是这一规则的例外，不从令牌端点获取令牌。

图 16-6 所示。

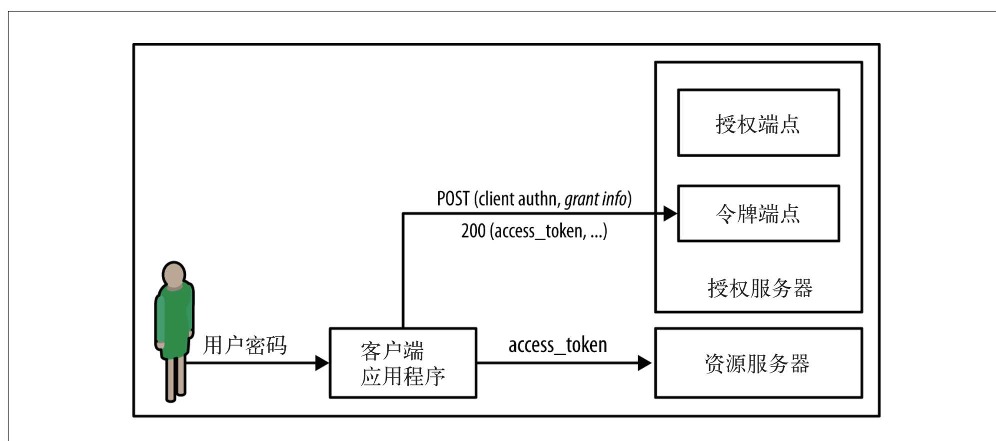


图 16-6：基于用户的密码凭据获得访问令牌

在 OAuth 2.0 框架中，这种情况称为*资源所有者密码凭据授予*（resource owner password credentials grant）流。初看起来，OAuth 2.0 提供这种支持似乎并不合理，因为其目标之一正是避免这种凭据泄露。然而，在某些场景中，这种方式是合理的，尤其是当用户在客户端应用程序中具有很高的信任度时（如企业应用场景）。首先，在 OAuth 2.0 中，客户端应用程序并不需要保存密码，用于每个请求，而只是将密码用于请求访问令牌，然后便可将密码立即移除。因此，如果用户修改了密码，访问令牌可能还是有效的。这种权限授予类型的另一个优点是实现较为简单，特别是当客户端是本地移动应用程序时。

另一个方法是使用*授权码*，表示用户无需使用密码即可执行的委托授权。这种方法称为*授权码授予*（authorization code grant）流，将在下一节介绍。

要获得令牌，我们可以向令牌端点 URI 发送一个 `POST` 请求，请求正文为类型为 `application/x-www-form-urlencoded`，其中包含权限授予类型及其值。例如，对授权码授予流，授予类型为 `authorization_code`，授予值为授权码。

```
POST https://authzserver.example/token_endpoint HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: authzserver.example

grant_type=authorization_code&
code=the.authorization.code
```

如果客户端是保密的（客户但持有分配的 `client_secret`），那么这个令牌请求还必须包含客户端身份验证信息。OAuth 2.0 框架定义了两种提供身份验证信息的方式：

- 使用 BasicHTTP 身份验证方案，其中 `client_id` 和 `client_secret` 分别用做用户名和密码；
- 将 `client_id` 和 `client_secret` 作为字段插入令牌请求正文。

如果令牌请求成功，那么响应消息会带有 application/json 正文，其中包含访问令牌的值、类型（例如：bearer）和有效期。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Cache-Control: private, max-age=0, must-revalidate

{"access_token":"the.access.token","token_type":"bearer", "expires_in":3600,
        ... other info ...}
```

16.4　授权码授予

使用授权码授予，用户不向客户端提供自己的身份信息，就可以将受限的授权委托给一个客户端应用程序。用户通过一个用户代理（例如：Web 浏览器或 Web View），直接与授权服务器的授权端点进行交互。授权码授予流的第一步是客户端应用程序将用户代理重定向到授权端点（参见图 16-7）。客户端使用授权端点请求 URI 的查询字符串，嵌入一组授权请求参数：

```
https://authzserver.example/authorization_endpoint?
  client_id=the.client.id&
  scope=user+repo&
  state=crCMc3d0acGdDiNnXJigpQ%3d%3d&
  response_type=code&
  redirect_uri=https%3a%2f%2fclient.example%2fcallback&
```

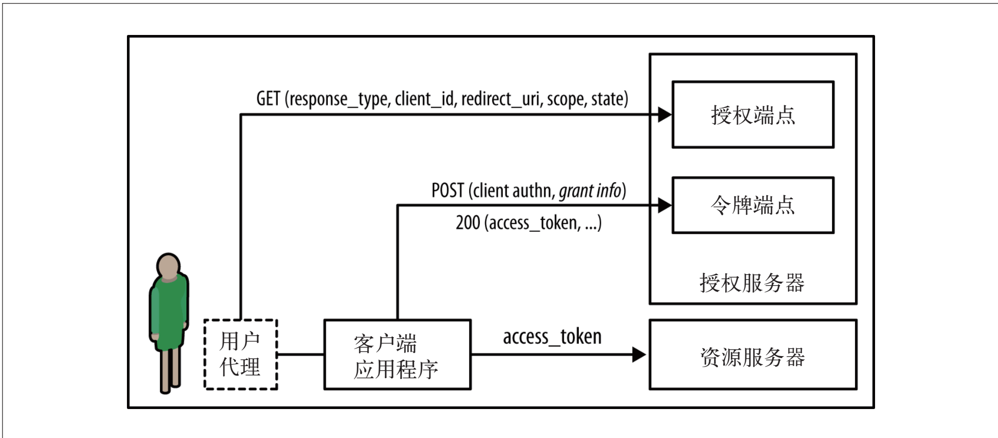


图 16-7：从授权服务器的授权端点请求授权授予

例如，参数 response_type 定义了要请求的权限授予，因为授权端点可以用于不同的令牌获取流；参数 scope 描述了要请求的授权特征。在收到请求后，授权服务器开始进行协议外（out-of-protocol）用户交互，目的是对用户进行身份验证，可能还要获取用户对客户端所请求授权的同意。

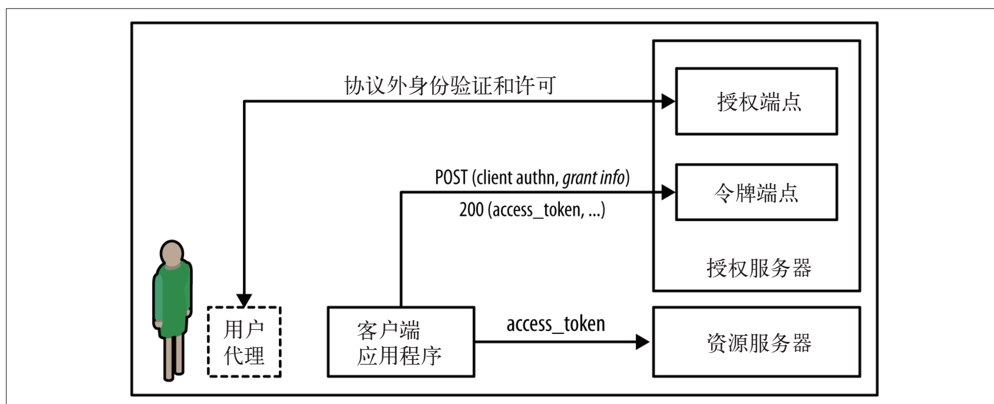


图 16-8：用户与授权端点之间的直接交互，以进行身份验证和授权许可

OAuth 2.0 协议没有定义用户身份验证协议，授权服务器可以自由选择最合适的身份验证协议，既可以使用简单的基于表单的用户名和密码方案，也可以使用分布式联邦协议。

在成功进行身份验证之后，授权服务器还可以询问用户，是否同意客户端应用程序请求的授权。在这里，授权服务器会使用客户端在注册过程中提供的描述信息（例如：应用程序名称、商标和主 URI），为用户提供更多信息。最后，授权服务器使用请求参数 `redirect_uri` 的值，将授权码嵌入请求 URI，为用户重定向到客户端应用程序。流程如图 16-9 所示，重定向 URI 如下所示：

```
https://client.example/callback?
  code=52...e4&
  state=cr...3D
```

出于安全考虑，客户端使用的重定向 URI 集合应该预先进行配置。T.AS 正是这样实现的，从图 16-3 中可以看到，每个客户端都关联到一组重定向 URI。

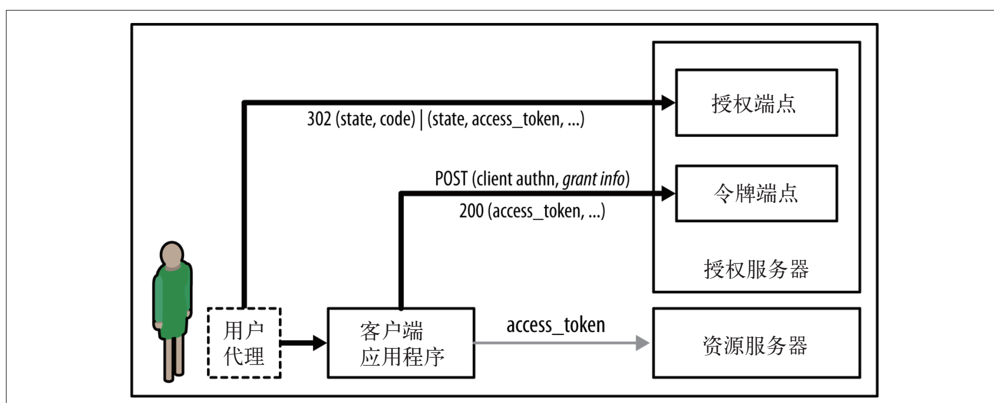


图 16-9：包含授权授予的授权端点响应

OAuth 2.0 框架定义的最后一个令牌获取流是*隐式赋予*（implicit grant），隐式赋予直接在授权端点的响应中返回访问令牌。这是唯一一个不从令牌端点返回访问令牌的 OAuth 2.0 流。

> **OAuth 2.0 授权码流示例**
>
> 位于 https://github.com/webapibook 的 OAuth2.Demos.AuthzCodeGrant 程序库，提供一个自托管的 OAuth 2.0 客户端控制台应用程序，其中使用了授权码流。这个应用程序预先配置为使用 GitHub API v3，但是也可以使用其他授权服务器和资源服务器。要使用 GitHub API 运行这个示例程序，你可以在 GitHub 的“account settings”中 provision 一个“developer application”，在代码中为 `client_id` 和 `client_secret` 赋值。这个示例受到的评价颇高，而且可以用于捕获 OAuth 2.0 协议消息。

16.5 范围

我们前面介绍过，OAuth 2.0 的目标之一是，支持客户端使用最终由用户委托的*受限授权*，进行资源访问。为此，OAuth 2.0 框架使用*范围*（scope）这个概念，定义这些授权的限制条件。范围的正式定义是一列以空格分隔的标识符，每个标识符定义一个授权类型。例如，GitHub Web API 使用的一些范围定义符有：

- `user` 授权客户端进行用户个人资料信息的读 / 写操作；
- `user:email` 授权客户端读取用户的电子邮件；
- `public_repo`：授权客户端进行用户的公共存储库的读 / 写操作。

因此，字符串 `user:email public_repo` 定义了一个范围，具有读取电子邮件和读 / 写存储库的授权。

通常情况下，这些范围标识符及其相关语义是由资源服务器定义的。范围标识符通常还具有相关的人工可读的描述，在向用户展示授权许可表格时使用。

举个例子，T.AS 中的范围由如下字段定义（参见图 16-3）：

- 范围标识符；
- 范围的显示名称和描述，用于与人类交互，例如请求获得用户的授权许可；
- 允许请求这一范围的一列客户端。

一个客户端可以使用的范围可能是受限的，如图 16-3 所示。

范围在协议中广泛使用，通常通过一个 `scope` 参数传入。当使用客户端凭证或资源所有者密码授予时，客户端可以在发送给令牌端点的令牌请求中包括这个 `scope` 参数，以定义请求授权。与此类似，当使用授权码或者隐式授权流时，客户端可以在发送给授权端点的授

权请求中包括 scope 参数。授权服务器可以基于用户的同意，返回一个不同于所请求的授权范围。因此，令牌响应中也可以包含 scope 参数，以通知客户端所授予的授权。

16.6 前通道与后通道

要更好地理解 OAuth 2.0 框架，我们需要认识到，客户端以两种不同的方式与授权服务器进行通信：*后通道*（back channel）方式和*前通道*（front channel）方式。*后通道*是客户端与令牌端点之间的直接通信，如图 6-5 所示。而前通道是客户端与授权端点之间，通过用户代理，基于 HTTP 重定向，进行间接通信（参见图 16-7）。因此，前通道具有一些明显的局限性。前通道基于重定向，因此对可以使用的 HTTP 功能有所限制：请求方法必须是 GET，请求信息必须位于请求 URI 中，即 URI 的查询字符串。

```
https://authz_server.example/authorization_endpoint?
  client_id=the.client.id&
  scope=user+repo&
  state=crCMc3d0acGdDiNnXJigpQ%3d%3d&
  response_type=code&
  redirect_uri=https%3a%2f%2fclient.example%2fcallback&
```

此外，响应永远是一个重定向，因此响应信息也要在重定向请求 URI 中传递：

```
https://client.example/callback?
  code=52...e4&
  state=cr...3D
```

如果出现错误，我们不能使用标准的 HTTP 状态码（响应永远是一个重定向），而是在 URI 的查询字符串中使用 error 和 error_description 参数，以此传递错误信息：

```
https://client.example/callback?
  error=access_denied&
  error_description=authorization+not+granted
```

而且，前通道通过用户代理运行，如果传输客户端凭证（client_secret），会对用户可见，因此无法安全传输客户端凭证。因此，在所有前通道请求中，客户端会进行标识（发送 client_id），但是不会进行身份验证。client_id 是公开的信息，攻击者很容易伪造出有效的授权请求。

最后，前通道还无法确保客户端发送给授权服务器的请求，和相应的响应之间的相关性，容易受到 CSRF（Cross-Site Request Forgery，跨站请求伪造）攻击。在跨站请求伪造攻击中，一个第三方恶意站点会让用户的浏览器向客户端发送一个请求，模拟一个 OAuth 2.0 前通道响应。使用这种技术，攻击站点可以控制客户端使用的访问令牌——例如，使用分发给攻击者帐户的授权码。

为了解决这个问题，OAuth 2.0 框架在请求和响应中都使用了一个状态参数，以确保其

相关性。客户端创建一个随机的状态值，将这个值包含在通过前通道发送的请求中。最后，授权服务器会在通过前通道返回的响应中包含同样的状态值。通过这种机制，客户端可以把收到的响应与之前发送的请求联系起来。RFC 6819——OAuth 2.0 Threat Model and Security Consideration——提供了关于如何有效使用这种保护机制的更多信息。

考虑到这些限制和问题，你可能会奇怪，为什么人们还在使用前通道。主要原因是，通过前通道，授权服务器可以和用户直接进行交互，不需要任何客户端的干预和出现。图 16-8 展示了授权服务器如何使用这一功能，对用户进行身份验证，并获得用户的授权许可。

另一方面，后通道将客户端与令牌端点直接连接，二者之间的通信可以不限于 HTTP 重定向。例如，客户端的令牌请求是一个 HTTP 的 `POST` 请求，参数位于请求正文中，响应可能使用 HTTP 状态码代表不同的错误条件（例如，`400` 表示错误请求，或者 `401` 表示客户端身份验证失败）。

如果客户端持有 `client_secret`（保密客户端），就必须在后通道中使用 `client_secret`，对自己进行身份验证。对此，OAuth 2.0 框架推荐使用 HTTP 基础认证方案，将用户名和密码分别替换为 `client_id` 和 `client_secret`：

```
POST https://authz_server.example/token_endpoint HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Authorization: Basic dGhlLmNsaWVudC5pZDp0aGUuY2xpZW50LnNlY3JldA==
Host: authz_server.example

grant_type=authorization_code&
code=the.authorization.code
```

后通道和前通道

其实 OAuth 2.0 RFC 文档并没有使用前通道和后通道这两个词。我们从 SAML 术语（参见 https://www.oasis-open.org/committees/download.php/21111/saml-glossary-2.0-os.html）中借用了这两个词，因为这两个词可以很好地帮助我们描述客户端与授权服务器进行通信的不同方式。

16.7 刷新令牌

访问令牌属于敏感信息，其使用生存期应该受到限制。为了解决这个问题，OAuth 2.0 框架定义了刷新令牌，用于获取新的访问令牌。当客户端向令牌端点发出请求，使用权限授予交换访问令牌时，得到的响应也可能包含一个刷新令牌。

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Cache-Control: private, max-age=0, must-revalidate

{"access_token":"the.access.token","token_type":"bearer", "expires_in":3600,
```

```
 "refresh_token":"the.refresh.token"}
```

客户端应用程序可以使用这个刷新令牌，获取新的访问令牌——例如，当旧令牌即将过期时。这一操作也在令牌端点完成，请求中的 grant_type 字段包含 refresh_token 的值：

```
POST https://authzserver.example/token_endpoint HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Host: authzserver.example

grant_type=refresh_token&
refresh_token=the.refresh.token
```

成功的响应将包含新的访问令牌和令牌生存期，还可以包含一个新的刷新令牌。

使用刷新令牌对客户端应用程序提出了更多的需求，客户端应用程序必须安全存储这一新信息，监控访问令牌的生存期，定期对令牌进行更新。但是，刷新令牌具有一些有用的属性。从安全的角度来看，缩短访问令牌的生存期，限制了恶意访问这一信息可能造成的后果。

从实现和优化的角度看，使用刷新令牌，系统可以采取混合方式，刷新令牌使用可撤销的生成物令牌，指向存储库中的一个条目；而访问令牌使用短时效的不可撤销的状态断言。采用这种方式，访问令牌的验证不需要访问存储库，更容易进行扩展。虽然访问令牌是不可撤销的，但是其缩短的生存期弥补了这一缺陷。另一方面，你可以在存储库中删除相关条目或将其标记为无效，很容易地撤销刷新令牌。

16.8 资源服务器和授权服务器

OAuth 2.0 框架明确定指出了服务器端的两个职责：

- *资源服务器*（resource server）提供访问受保护资源的接口，是访问令牌的使用者；
- *授权服务器*（authorization server）的职责众多，其中之一是颁发访问令牌，用于访问受保护的资源。

这并不是说资源服务器和授权服务器必须是两个独立的实体。这两个角色完全可以由同一个物理机器以及同样的软件组件实现。但是，OAuth 2.0 框架的确支持分离的架构，其中授权服务器由单独的软件组件运行（例如：Thinktecture 授权服务器）。资源服务器甚至可以依赖由另一个实体运行的外部授权服务器（例如：Windows Azure Active Directory）。

虽然支持这些分离的架构，但是 OAuth 2.0 框架并没有指定独立的资源服务器和授权服务器应当如何协作。OAuth 2.0 框架对于一些方面（如访问令牌格式和验证过程）未加定义，我们必须针对每个场景进行定义。请注意，从用户或客户端的角度，这些具体实现不会产生任何影响，因为访问令牌本来就对用户或客户端不透明。

另外，OAuth 2.0 框架也没有定义，授权服务器通过访问令牌要传递给资源服务器哪些信息。最直接的做法是，用访问令牌表示令牌请求和相关授权，包括的信息有：

- 资源所有者标识（如果客户端不是自发进行访问）；
- 发起请求的客户端，由其 client_id 标识；
- 所请求的授权范围，由 scope 字符串标识。

与访问令牌相关的信息还应该包括其时间有效期（令牌并非永久有效）以及令牌的受众，也就是令牌发往的资源服务器的一些标识信息。

例如，T.AS 使用 JWT 格式标识访问令牌。示例 16-1 展示了这种 JWT 令牌的有效载荷，其中有：

- sub 声明（其中包含用户唯一标识符）和 role 声明（其中包含附加的用户声明）；
- client_id 声明，其中包含客户端应用程序标识；
- scope 声明，其中包含所授予的授权范围。

令牌有效载荷还包含令牌颁发者的标识（iss 声明），以及令牌针对的目标或受众（aud 声明）。

示例 16-1：由 Thinktecture 授权服务器颁发的一个访问令牌的 JWT 有效载荷

```
{
    "exp": 1379284015,
    "aud": "http://resourceserver.example",
    "iss": "http://authzserver.example",
    "role": [
            "fictional_character",
            "student"
    ],
    "client_id": "client2",
    "scope": [
        "scope1",
        "scope2"
    ],
    "nbf": 1379280415,
    "sub": "Alice"
}
```

与示例 16-1 中的 role 声明一样，用户标识也可以不限于简单的标识符，很好地契合了第 15 章中介绍的声明模型。

16.9　在ASP.NET Web API中处理访问令牌

我们在 15.3.8 节介绍过，Katana 项目提供一组身份认证中间件类，其中的 OAuthBearerAuthenticationMiddleware 实现了 OAuth 2.0 Bearer 认证方案。OAuthBearerAuthenticationMiddleware 的行为通过 OAuthBearerAuthenticationOptions 类进行配置（参见图 16-10）。当接收到一个请求时，相关的身份验证处理程序会执行以下步骤。

(1) 获取令牌。如果请求包含方案为 Bearer 的 Authorization 标头，那么使用该标头值作为访问令牌。否则，身份验证方法不返回任何身份信息。

(2) 获取身份认证票据。从消息获得令牌之后，Options.AccessTokenFormat.Unprotect 方法从访问令牌获取一个身份验证票据。正如我们之前看到的，这个票据既包含基于声明的

身份信息，也包含附加的身份验证属性。

(3) 验证身份验证票据。最后一步，检查身份验证票据的有效性。

在处理程序最后，如果所有的步骤都执行成功，那么请求的访问令牌会转换为一个身份信息返回。通过 `Options.Provider` 和 `Options.AccessTokenProvider`，你可以对之前的步骤进行定制。例如：如果定义了 `Options.Provider`，你可以从消息的其他部分获取访问令牌，也可以验证和修改获取到的身份信息。

在默认情况下，Katana 使用一个定制的访问令牌格式，在其自己的授权服务器中使用。但是，通过修改 `Options.AccessTokenFormat`，你也可以对 Katana 进行配置，使其接受基于 JWT 的访问令牌。Katana 的 `UseJwtBearerAuthentication` 扩展方法就实现了这一功能，过程如下。

(1) 使用一个 `JwtBearerAuthenticationOptions` 参数，其中包含了如何验证 JWT 令牌的信息，其中有：允许的受众和签名验证信息。

(2) 在内部创建一个使用 `JwtFormat` 配置的 `OAuthBearerAuthenticationOptions`，`JwtFormat` 是一个使用 JWT 格式的 `ISecureDataFormat` 实现。

- 使用 `OAuthBearerAuthenticationOptions`，注册 `OAuthBearerAuthenticationMiddleware`。

图 16-10 展示了这一过程中用到的类。示例 16-2 展示了如何利用这一扩展方法，配置一个基于 Katana 的资源服务器，操作如下：

- 将 `AllowedAudiences` 属性配置为资源服务器的 URI；
- 将 `IssuerSecurityTokenProviders` 属性配置为授权服务器的对称签名密钥。

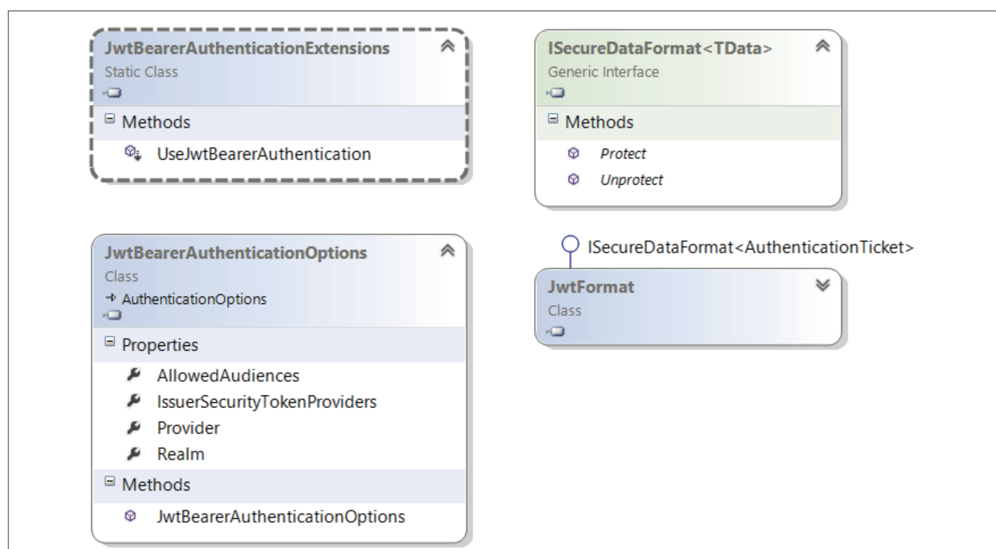


图 16-10：使用基于 JWT 的访问令牌的类

示例 16-2：将一个基于 Katana 的资源服务器配置为使用 T.AS

```
config.Filters.Add(new HostAuthenticationFilter("Bearer"));

app.UseJwtBearerAuthentication(new JwtBearerAuthenticationOptions
{
    AllowedAudiences = new []
    {
        "http://resourceserver.example"
    },
    IssuerSecurityTokenProviders = new []
    {
        new SymmetricKeyIssuerSecurityTokenProvider(
                "http://authzserver.example",
                "the.authorization.symmetric.signature.key")
    },

    Realm = "resourceserver.example",

    AuthenticationMode = AuthenticationMode.Passive
});
```

16.10 OAuth 2.0与身份验证

从 RFC 6749 的文档名（OAuth 2.0 授权框架）就可以看出，OAuth 2.0 关注的主要内容是授权，而非身份验证。OAuth 2.0 的主要目标是，支持客户端应用程序以自发或代表用户的方式，访问 Web API 提供的资源子集。但是，OAuth 2.0 框架的实现也可以提供某些形式的身份验证功能。

正如我们在本章开头介绍的，由客户端应用程序向资源服务器发起的请求，包含一个访问令牌。这个令牌的主要目的是向资源服务器证明，令牌的持有者（即发起请求的客户端应用程序）得到了用户的授权，可以访问受保护的资源。要实现这一目的，通常的做法是使用访问令牌完成如下功能：

- 对发送请求的客户端应用程序进行身份验证；
- 对进行授权的用户进行身份验证；
- 定义授权范围。

图 16-11 描绘了这一身份验证场景，其中授权服务器是身份信息提供方，资源服务器是转发方，客户端和用户都是身份信息的主体。例如，T.AS 颁发的 JWT 令牌（参见示例 16-1）恰好包含这三个信息：`client_id`、用户声明（`role` 和 `sub`）和授权范围。还需要注意的是，使用 Katana 中间件时，访问令牌信息会转换为一个用户身份对象，传递到上层。但是，我们前面介绍过，OAuth 2.0 框架没有定义访问令牌的格式和信息，由具体实现决定这些细节。如果在一个 OAuth 2.0 的实现中，访问令牌只包含授权资源和 HTTP 方法，而没有任何关于客户端或用户的信息，也是完全可能的。因此，在大多数时候访问令牌也是

一个身份验证令牌，向资源服务器提供客户端和用户的身份验证信息，但是这要取决于框架的具体实现。

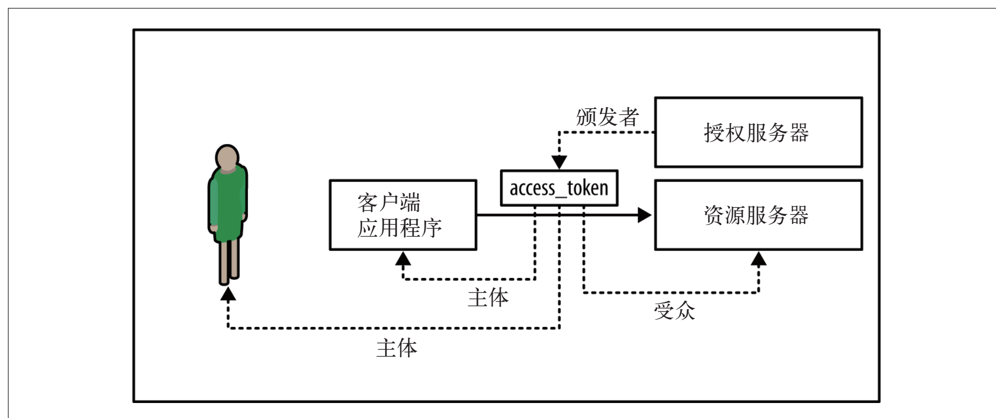


图 16-11：使用访问令牌向资源服务器提供客户端和用户的身份验证

OAuth 2.0 框架也可以用作另一种身份验证的基础：向客户端应用程序提供用户的身份验证。客户端可以使用包含用户身份信息的上下文相关资源（称作*用户信息资源*），对用户进行身份验证。例如，在 GitHub API v3 中，对 https://api.github.com/user 的成功 `GET` 操作会返回一个资源表示，其中包含客户端所使用的访问令牌代表的用户的姓名和电子邮件地址。客户端应用程序可以借此获得资源服务器和授权服务器宣称的用户身份信息（参见图 16-12）。

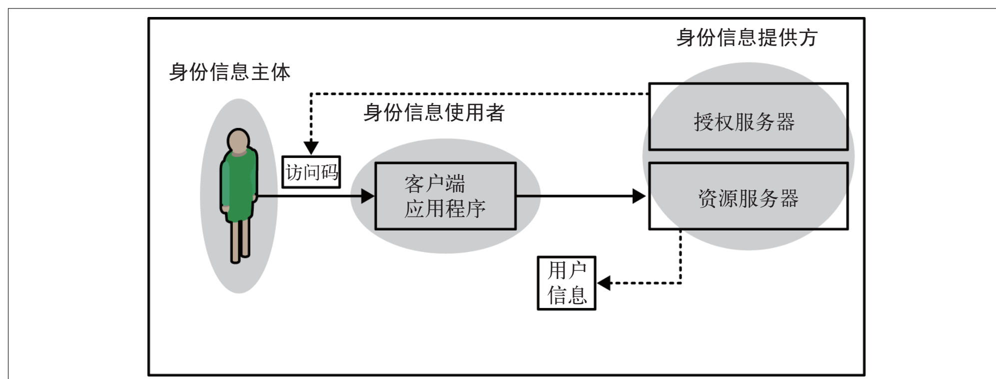


图 16-12：客户端使用受保护资源验证用户身份

在我们编写这本书时，已经经常可以看到 Web 应用程序使用这种基于 OAuth 2.0 的技术，向社交身份信息提供方（例如：Facebook 或 GitHub）验证用户的身份。但是，这种做法存在两个缺点。首先，这种做法依赖上下文相关的*用户信息资源*，而这种资源并未在 OAuth 2.0 框架中定义。这意味着我们必须针对每个不同的资源服务器进行定制。其次，最重要的是，这种身份验证用法不是对所有 OAuth 2.0 框架流都安全——具体来说，对使用公共

客户端的隐式流和授权码流不安全。例如，在一些流中，客户端应用程序可以使用授权码或者令牌，把自己作为用户，在另外一个应用程序中进行身份验证。在隐式流中，用户代理直接把令牌从授权端点传递到客户端应用程序。在接收到令牌后，一个恶意客户端可以向另一个客户端提供这个令牌，将自己伪装成用户。当第二个客户端访问这个用户信息资源时，得到的将是原始用户的身份信息。

OpenID Connect（http://openid.net/connect/）规范在 OAuth 2.0 框架之上提供一个身份信息层，致力于解决这两个问题。[1] 首先，OpenID Connect 规范对用户信息资源（在这个规范中称为 UserInfo Endpoint）的概念及其返回的标识进行了标准化：一个成员为声明的 JSON 对象，OpenID Connect 也定义了这些声明的含义。示例 16-3 是 Google UserInfo 资源（位于 https://www.googleapis.com/oauth2/v3/userinfo）为我们虚构的用户 Alice 返回的声明。

示例 16-3：UserInfo 资源返回的表示

```
{
 "sub": "104107606523710296052",
 "email": "alice4demos@gmail.com",
 "email_verified": true
}
```

OpenID Connect 还扩展了 OAuth 2.0 的令牌响应定义，添加了一个 `id_token` 字段（参见示例 16-4）。`id_token` 字段的值是一个签名的 JWT 令牌，其中包含用户声明，可供客户端使用。请注意，这与访问令牌的概念相悖，访问令牌对客户端是不透明的。示例 16-5 展示了一个 ID 令牌的有效载荷，其中包含关于用户的身份信息声明（`email` 声明），但也包含该令牌针对的受众：`aud` 声明，`aud` 声明的值是客户端应用程序的 `client_id`。这个 `aud` 字段将一个 ID 令牌绑定到一个使用者，以防止恶意客户端在其他客户端重用这个令牌。

示例 16-4：包含 ID 令牌的令牌响应

```
{
  "access_token" : "ya..8s",
  "token_type" : "Bearer",
  "expires_in" : 3599,
  "id_token" : "ey..OQ"
}
```

示例 16-5：令牌响应中返回的 ID 令牌有效载荷

```
{
    "sub": "104107606523710296052",
    "iss": "accounts.google.com",
    "email_verified": "true",
    "at_hash": "G_...hQ",
    "exp": 1380480238,
    "azp": "55...ve.apps.googleusercontent.com",
    "iat": 1380476338,
```

注 1：虽然名为 OpenID Connect，但这个规范更类似 OAuth 2.0，而非经典的 OpenID 协议。

```
        "email": "alice4demos@gmail.com",
        "aud": "55...ve.apps.googleusercontent.com"
    }
```

> **OpenID Connect 授权码流示例**
>
> `WebApiBook.Security` 存储库位于 https://github.com/webapibook，包含一个自托管的 OpenID Connect 客户端控制台应用程序，其中使用了授权代码流。这个示例程序预先配置为使用谷歌授权服务器，但是也可以使用其他的 OpenID Connect 实现。

通过在 OAuth 2.0 之上添加一个身份信息层，OpenID Connect 为客户端应用程序提供了一个一致的协议，以完成以下功能：

- 获取一个签名的身份信息声明集合，对其用户进行身份验证；
- 获取一个访问令牌，使客户端可以访问代表用户访问受保护的资源。

请注意，经典的身份信息联邦协议（例如：SAML、WS-Federation 或经典的 OpenID 协议）只提供第一个功能。而 OAuth 2.0 框架只提供第二个功能。

16.11　基于范围的授权

使用 OAuth 2.0 框架时，你可以将授权决定从资源服务器中移走，在授权服务器中实现。我们之前介绍过，访问令牌有与其相关的范围，确切定义了将什么权限授予客户端。因此，在这种情况下，资源服务器就只需要执行令牌范围中定义的授权决定。

第 15 章中介绍的 `Thinktecture.IdentityModel.45` 程序库也提供一个定义范围的授权属性 `ScopeAttribute`。`ScopeAttribute` 的构造函数参数为一列范围标识符，只有当相关联的声明用户对象的范围声明与这些标识符全部匹配时，请求才能得到授权。

示例 16-6 展示了 `ScopeAttribute` 的用法，其中 `POST` 请求要求使用带有 `create` 范围标识符的访问令牌，而 `DELETE` 请求要求使用带有 `delete` 标识符的访问令牌。

示例 16-6：使用 `ScopeAttribute`

```
    public class ScopeExampleResourceController : ApiController
    {
        public HttpResponseMessage Get()
        {
            return new HttpResponseMessage
            {
                Content = new StringContent("resource representation")
            };
        }

        [Scope("create")]
```

```
        public HttpResponseMessage Post()
        {
            return new HttpResponseMessage()
            {
                Content = new StringContent("result representation")
            };
        }

        [Scope("delete")]
        public HttpResponseMessage Delete(string id)
        {
            return new HttpResponseMessage(HttpStatusCode.NoContent);
        }
    }
```

16.12 小结

本章介绍了 OAuth 2.0 授权框架，包括其所用协议和模式。OAuth 2.0 授权框架有几个值得强调的功能。首先，OAuth 2.0 框架引入了一个模型，对用户、客户端和资源服务器进行了明确的区分。当这三个角色分属不同的信任域时，这种概念区分变得尤为重要，而且将极大影响我们对基于 Web 系统进行安全建模的方式。OAuth 2.0 还引入了授权服务器的概念，将其定义为一个实体，负责颁发和管理客户端（代表用户）用于访问资源的访问令牌。

在我们编写这本书时，大部分 OAuth 2.0 实现所使用的身份验证服务器，都与使用其服务的资源服务器相关联。但是，有一些项目，例如：Thinktecture 的授权服务器和 Windows Azure Active Directory，正在开始提供可以用在多种上下文，与不同资源服务器一起工作的授权服务器。OAuth 2.0 框架也提供了用于不同场景的具体模式和协议，既支持客户端自发访问资源（*客户端凭据授予流*），也支持通过前通道交互进行受限的授权委托（*授权码授予流*）。

OAuth 2.0 还是新的 OpenID Connect 协议的基础。OpenID Connect 提供了非集中式的身份验证功能。所有这些技术为我们提供了一种集成的方式，解决 Web API 中存在的一些身份验证和授权问题。

虽然 OAuth 2.0 框架目前得到了极大的支持，但是也受到了许多的批评。首先，OAuth 2.0 框架提供了许多的协议选项和替代方法，妨碍了互交互性。OAuth 2.0 框架的灵活性也对安全产生了负面影响：由于可选项过多，不安全实现的可能性也增加了。持有者令牌[1]的缺点（参见第 15 章）也为 OAuth 2.0 招致了批评。

尽管具有这些问题，OAuth 2.0 框架仍然是当前 Web API 安全领域的一个重要组成部分。

注 1：在我们编写本章时，持有者令牌是唯一完成的规范；MAC 规范仍未完成。

第17章

可测试性

解决问题的难处在于问题还会反弹。

作为开发者，我们经常发现自己花费了大量时间，尝试解决 Web API 实现中可能发生的问题。在很多时候，我们使用浏览器或者 HTTP 调试程序，用试错方法进行测试，但是这种手工测试非常耗费时间，无法重复，而且容易出错。雪上加霜的是，随着我们的 Web PI 覆盖的使用场景数量增加，手工测试每个可能的执行路由变成了一项令人望而生畏的任务。

多年以来，业内出现了各种工具和实践，帮助开发者的工作。例如：自动化测试领域就得到了许多的改进。我们在这里所说的自动化测试，是指创建或配置一个软件，为我们执行测试。采用自动化测试具有很多显而易见的优点：测试可重复，我们可以在任何时间运行测试，甚至可以定时自动运行测试，无需进行任何的交互。

在本章中，我们将讨论开发者对 ASP.NET Web API 进行自动化测试的两个最常见的选择：单元测试和集成测试。对于不熟悉这些概念的读者，我们简要介绍了自动化测试，还提到了 TDD（Test-Driven Development，测试驱动开发）这一核心实践。自动化测试涉及的范围很广，因此我们的讨论只限于 ASP.NET Web API 的测试，以及如何使用这一技术，测试使用 ASP.NET Web API 框架构建的所有组件。

17.1 单元测试

为了验证其他某些代码单独运行时的预期行为，我们通常会编写一些代码，这些代码就是单元测试。编写单元测试时，我们首先将应用程序代码分为离散的部分，如类的方法，这些部分容易管理，可以单独进行测试而不互相影响。例如，对于 ASP.NET Web API，你可

能想为 `ApiController` 的一个具体实现的每个公共方法编写一个单元测试。"单独"意味着测试之间没有依赖关系，因此测试运行的顺序不应该影响测试的最终结果。实际上，你应该能够同时运行所有的单元测试。

一个单元测试通常分为如下三部分。

(1) 准备（arrange）：将一个或多个对象设置为已知的状态。
(2) 操作（act）：对这些对象的状态进行操作，例如：调用对象的方法。
(3) 断言（assert）：检查测试的结果，将其与预期结果进行比较。

和源代码一样，我们也应该将单元测试作为开发的生成物，将其保存在源代码库中。将测试保存在代码库中，你就可以将测试用于多种用途，例如，记录某段代码的预期行为，或者在实现发生变化时确保代码行为依然正确。单元测试应该容易运行，并且运行速度很快。否则，开发者不太可能会运行这些测试。

17.1.1 使用测试框架

单元测试框架强制实现某些测试结构，帮助我们简化编写单元测试的过程，并提供运行这些测试的工具。和任何框架一样，单元测试框架不是必须的，但的确可以加快实现和运行单元测试的过程。

今天，开发者最常用的单元测试框架通常是 xUnit 家族的一员，其中有：Visual Studio 单元测试工具（Visual Studio 的付费版本中提供），或开源项目 xUnit.NET。本章，我们将使用 xUnit.NET 进行集成测试和单元测试的讨论。xUnit 家族的大部分框架要么是 JUnit 的直接移植，要么是借用了 JUnit 的一些概念或想法，这些概念和想法最初出现在极限编程中，并逐渐流行起来。

17.1.2 Visual Studio单元测试入门

为了使开发者的工作更容易，ASP.NET 团队在 Visual Studio 中，ASP.NET Web API 应用程序的 New Project 对话框中，提供了单元测试支持（参见图 17-1）。

通过选中 Create Unit Test project 复选框，你就告知了 Visual Studio project 向导，准备使用你偏好的测试框架（默认情况下，Visual Studio 单元测试工具是唯一可用的框架）创建一个新的单元测试项目。当你选择 Visual Studio Unit Testing 时，Visual Studio 还会生成一个项目，其中包含一组单元测试，测试默认模板中的 ASP.NET Web API 控制器。对于其他的测试工具，根据该工具在 Visual Studio 中注册的项目模板定义，Visual Studio 的行为会发生改变。

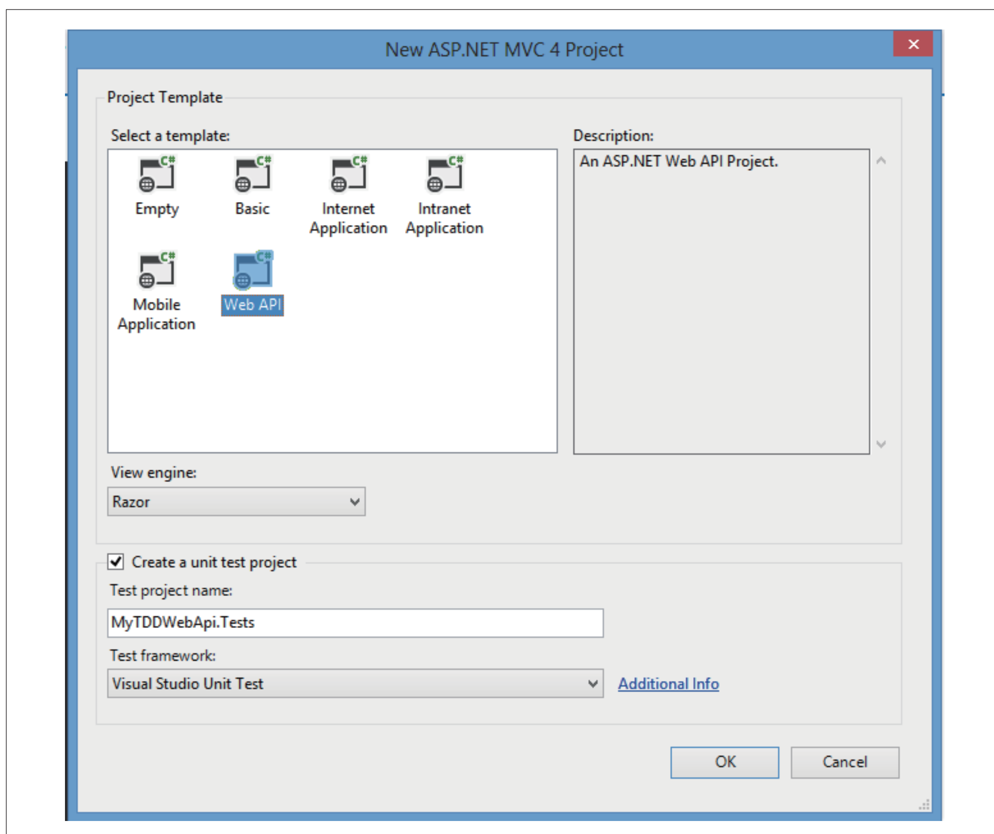


图 17-1：New Project 对话框

对于 ASP.NET Web API 项目模板，因为该模板包含一个 `ValueController`，你会在测试项目中找到对应的单元测试 `ValueControllerTest`。示例 17-1 列出了 Visual Studio 在 `ValueControllerTest` 中生成的 `Get` 方法。

示例 17-1：ASP.NET Web API 项目模板生成的单元测试

```
[TestClass]
public class ValuesControllerTest
{
  [TestMethod]
  public void Get()
  {
    // 准备
    ValuesController controller = new ValuesController(); // <1>

    // 操作
    IEnumerable<string> result = controller.Get(); // <2>

    // 断言 // <3>
    Assert.IsNotNull(result);
```

```
        Assert.AreEqual(2, result.Count());
        Assert.AreEqual("value1", result.ElementAt(0));
        Assert.AreEqual("value2", result.ElementAt(1));
        }
    }
```

你可以看到，和我们之前介绍的一样，这个方法分为准备、操作和断言三部分。在准备部分 <1>，待测试的 ValueController 进行了初始化，然后在操作部分 <2> 调用了 Get 方法，断言部分 <3> 对预期值进行了断言。

Assert 类，以及这个单元测试中用到的 TestClass 和 TestMethod 属性，都是 Visual Studio 单元测试框架的一部分。在 xUnit 家族的任何框架中，你通常都能找到这三个类 / 属性，虽然名字可能不同，但功能是类似的。

示例 17-1 展示了如何对某个特定的控制器进行单元测试，但是你也会对其他组件编写单元测试，例如：封装数据访问或业务逻辑的组件，以及 Web API 中才有的组件（如消息处理程序）。

17.1.3 xUnit.NET

xUnit.NET 也是 xUnit 家族中的一员，最初是由 Brad Wilson 和 James Newkirk 提出的一个开源计划。James Newkirk 也是 JUnit 的第一个 .NET 平台移植，NUnit 的作者之一。xUnit.NET 的设计目标是将从以往经验中得到的很多最佳实践以及教训，应用在单元测试中，更好地适应 .NET 平台的最新变化。例如：xUnit.NET 框架提供的在测试中检验异常的方法，比其他传统框架的做法更为优雅。虽然大部分框架使用属性来处理异常检验，但是 xUnit.NET 使用了委托，如示例 17-2 所示。

示例 17-2：检查预期异常的单元测试

```
[TestClass]
public class ValuesControllerTest
{
  [TestMethod]
  public void Get()
  {
    // 准备
    ValuesController controller = new ValuesController();

    // 断言 // <1>
    controller.Throws<HttpException>(() => controller.Get("bad")) // <2>
  }
}
```

你可以看到，Throws 方法提供一个简单的方式，在一个代码行中表达同时表达断言 <1> 和操作 <2>，将委托传递给负责抛出异常的方法。

1. 单元测试的组织

一般来说，一个单元测试应该只测试一个功能或行为，否则很难得到关于具体错误的真实反馈。此外，单元测试自身不应该提供任何的价值，否则很难判断预期的行为应该是什么。因此，单元测试通常是以提供细粒度反馈的方法组织的。每个方法应该只展示一种预期的行为，但是可能会用到一个或多个断言。在 xUnit.NET 中，这些方法必须带有 `Fact` 属性，以标识自己为单元测试。你可能还想使用不同的标准，将单元测试组织为群组，例如：测试某个组件的单元测试，或者测试某个具体用例的单元测试。xUnit.NET 使用类将所有的测试组织在一起，形成单元测试术语中通常所说的*测试集*（test suite）。

2. `Assert`类

大多数的 xUnit 框架，包括 xUnit.net，都使用一个提供常用静态方法的 `Assert` 类，使用流接口（fluent interface）进行比较或者检验，流接口可以更明确地表达操作意图。例如，如果要验证一个方法调用的返回值应该不为 `null`，那么 `Assert.IsNotNull(result)` 可能比 `Assert.IsTure(result == null)` 更能表达你的意图。但这只是编写测试的开发者的个人偏好而已。当条件值为 `false` 时，`Assert` 类的方法都会抛出异常，使我们可以得知一个单元测试是否失败了。

17.1.4 单元测试在测试驱动开发中的作用

TDD（测试驱动开发）是一种设计技术，要求你首先编写测试，然后实现所需的应用程序代码使得测试通过，从而以单元测试驱动产品代码的设计。如果正确地应用了测试驱动开发，你得到的生成物将是应用程序代码，以及描述预期行为的单元测试。之后你可以随时使用这些单元测试，确保产品代码的行为依然正确。但是，测试驱动开发不仅可以使用单元测试减少产品代码中的缺陷，而且可以改进代码的设计。通过首先编写测试，你在产品代码实际编写之前就描述了其预期行为。你编写的是你需要的产品代码，没有机会编写任何不必需要的实现。

人们常犯的一个错误是，认为编写一些单元就是使用了测试驱动开发。虽然测试驱动开发要求使用单元测试驱动代码设计，但是编写单元测试并不一定就是测试驱动开发。你可以在编写代码之后编写单元测试，这通常是为了提高测试的代码覆盖率，但是这并不意味着你使用了测试驱动开发，因为产品代码在单元测试之前就已经存在了。

1. 红绿周期

在编写单元测试时，“*红*”和“*绿*”可以分别用于替代*失败*和*成功*。大多数的测试运行器使用这两个颜色，帮助开发者快速识别哪些测试通过或失败。测试驱动开发也大量使用这两种颜色，驱动新功能的开发。因为你测试的代码还不存在，所以首次运行测试会失败。在编写测试通过所需的代码后重新运行测试，如果产品代码的行为正确，你将会得到一个绿灯；如果产品代码还需改进，你将会得到一个红灯。这样一来，你会不断地经过红 / 绿

周期。但是，一旦一个测试通过，你就应该停止编写新的产品代码，直到测试新需求的新测试失败。基本上，你编写足够的应用程序代码使测试通过，这种做法初看似乎并不完美，但是你拥有了一个工具，可以验证对产品代码的任何改进都不会影响预期的行为。如果你想对产品代码的某些方面进行改进或优化，那就可以放手去做，只要不修改组件的公共接口即可，还可以使用原有的测试确保底层行为依然不变。

2. 代码重构

*代码重构*是指修改组件内部实现而保持其可见行为不变的操作。例如，将一个大型的公共方法改成一组功能单一，更易维护的小型方法。在重构产品代码时，你不能修改已有的单元测试。修改单元测试意味着增加、删除或修改已有的功能。已有的单元测试可以用来确保，在重构之后，应用程序代码的可见行为没有改变。举个例子，如果你有一个控制器操作，返回客户列表，并编写了验证这个行为的单元测试。不管获得列表的内部实现如何，这个单元测试只预期得到同样的客户列表。如果你在代码中发现了一些问题，就可以进行重构，改进代码，并使用已有的单元测试，确保新加入的修改不会破坏任何功能。

示例 17-3 展示了一些可以通过内部重构进行改进的代码。

示例 17-3：两个实例化 `HttpClient` 的方法

```
public abstract class IssueSource : IIssueSource
{
  HttpMessageHandler _handler = null;

  protected IssueSource(HttpMessageHandler handler = null)
  {
    _handler = handler;
  }

  public virtual Task<IEnumerable<Issue>> FindAsync()
  {
    HttpClient client;

    if (_handler != null)
      client = new HttpClient(_handler);
    else
      client = new HttpClient();

    // 使用 HttpClient 实例……
  }

  public virtual Task<IEnumerable<Issue>> FindAsyncQuery(dynamic values)
  {
    HttpClient client;

    if (_handler != null)
      client = new HttpClient(_handler);
    else
      client = new HttpClient();

    // 使用 HttpClient 实例……
```

```
    }
  }
```

这两个方法都创建了一个新的 HttpClient 实例。如果实例化代码需要进行修改，如添加新的设置，那么这两个方法都需要修改。为此，我们可以将实例化代码移到一个通用的内部方法中（参见示例 17-4）。

示例 17-4：HttpClient 实例化代码移到通用方法中

```
  public abstract class IssueSource : IIssueSource
  {
    HttpMessageHandler _handler = null;

    protected IssueSource(HttpMessageHandler handler = null)
    {
      _handler = handler;
    }

    public virtual Task<IEnumerable<Issue>> FindAsync()
    {
      HttpClient client = GetClient();

      // 使用 HttpClient 实例……
    }

    public virtual Task<IEnumerable<Issue>> FindAsyncQuery(dynamic values)
    {
      HttpClient client = GetClient();

      // 使用 HttpClient 实例……
    }

    protected HttpClient GetClient()
    {
      HttpClient client;

      if (_handler != null)
        client = new HttpClient(_handler);
      else
        client = new HttpClient();

      return client;
    }
  }
```

我们移除了重复代码，但没有修改这个类的外部接口。预期的行为保持不变，因此单元测试也无需修改，可以用于验证我们的改动没有破坏任何功能。

3. 依赖注入和模拟

在静态语言的单元测试中，因为依赖项无法在运行时很容易地替换，所以经常会使用依赖注入。单元测试关注的是测试特定代码在隔离环境中的行为，因此需要尽量减少任何外部依赖项在测试时的影响。使用依赖注入，你可以对所有硬编码的依赖项进行替换，在运行时注入

依赖，以伪造依赖项的行为。例如，如果一个 Web API 控制器依赖一个数据访问类进行数据库查询，那么我们不希望对带有这一明确依赖的控制器进行单元测试，因为每次测试运行时都需要初始化和准备好数据库。我们首先需要利用接口或抽象类，把控制器和实际访问数据库的类实现分开，然后通过构造函数参数或者属性设置方法，将其注入控制器。剩下的任务就是由单元测试创建一个伪造的类，这个类实现之前用到的接口或抽象类，同时满足测试的需求。这个伪造的类可以简单地模拟相同的接口，使用在这个测试中预先初始化的内存列表，为测试返回预期的值。使用这种方法，我们可以去除测试中的数据库依赖。我们可以使用多个开源框架，例如 Moq 或 RhinoMocks，从一个接口或基类自动生成伪造的或模拟的类，并设置这个模拟类的预期行为。示例 17-5 是我们的问题跟踪 Web API 的一段单元测试，测试中实例化了一个模拟类（由 Moq 框架生成），以模仿数据访问类的行为。

示例 17-5：使用模拟类的单元测试

```
public class IssuesControllerTests
{
  private Mock<IIssueSource> _mockIssueSource = new Mock<IIssueSource>(); // <1>
  private IssuesController _controller;

  public IssuesControllerTests()
  {
    _controller = new IssuesController(_mockIssueSource.Object); // <2>
  }

  [Fact]
  public void ShouldCallFindAsyncWhenGETForAllIssues() {
    _controller.Get();
    _mockIssueSource.Verify(i=>i.FindAsync()); // <3>
  }
}
```

示例代码使用 Moq 框架提供的 `Mock` 类，实例化了 `IIssueSource` 接口的一个模拟类 <1>，并把这个实例注入 `IssuesController` 构造函数 <2>。这个单元测调用了控制器的 `Get` 方法，并验证 `Mock` 对象的 `FindAsync` 方法实际得到了调用 <3>。`Verify` 方法也是由 Moq 框架提供的，用于检验一个方法是否得到调用，以及调用次数（例如，调用 `FindAsync` 方法超过一次就说明代码存在缺陷）。如果我们没有使用 Moq 框架，那么要实现类似的功能就需要手工编写重复的代码。

17.2　对ASP.NET Web API实现进行单元测试

ASP.NET Web API 实现中有一些组件需要隔离测试。在本章中，我们将介绍其中几个，例如：`ApiController`、`MediaTypeFormatter` 和 `HttpMessageHandler`。在本章最后，我们还会探讨如何使用 HttpClient 类和内存托管进行集成测试。

17.2.1　测试ApiController

在 ASP.NET Web API 中，`ApiController` 是你的 Web API 实现的入口，作为一个桥梁，通

过 HTTP 向外界提供应用系统的逻辑。ApiController 需要测试的主要的一点是，在隔离环境下，测试控制器如何对不同的请求消息做出反应。你可以根据 Web API 支持的场景或用例，选择使用哪种消息。隔离也是单元测试的关键因素，因为你无法假定在测试运行之前，Web API 运行时中的各种条件都正确进行了设置。例如，如果你的 ApiController 依赖通过身份验证的用户执行某种操作，那么就必须在单元测试中对用户进行配置。请求消息的初始化也是如此。

在这一节中，我们将使用之前构建的管理问题的 ApiController 作为起点，尝试为这个控制器编写单元测试，覆盖支持的一些用例。让我们从示例 17-6 开始。

示例 17-6：我们的 IssuesController 实现

```
public class IssuesController : ApiController
{
  private readonly IIssueSource _issueSource;

  public IssuesController(IIssueSource issueSource )
  {
    _issueSource = issueSource;
  }

  public async Task<Issue> Get(string id) // <1>
  {
    var issue = await _issueSource.FindAsync(id);
    if(issue == null)
      throw new HttpResponseException(HttpStatusCode.NotFound);
    return issue;
  }

  public async Task<HttpResponseMessage> Post(Issue issue) // <2>
  {
    var createdIssue = await _issueSource.CreateAsync(issue);
    var link = Url.Link("DefaultApi", new {Controller = "issues",
      id = createdIssue.Id});
    var response = Request.CreateResponse(HttpStatusCode.Created, createdIssue);
    response.Headers.Location = new Uri(link);
    return response;
  }
}
```

示例 17-6 是 IssuesController 的最初实现，初看起来并不复杂。IssuesController 包含一个 Get 方法，用于获取存在的问题 <1>，以及一个 Post 方法，用于添加新问题 <2>。IssuesController 还依赖一个 IIssueSource 实例进行持久性问题的处理。

1. 测试Get方法

我们的第一个 Get 方法（参见示例 17-6）看起来非常简单，这个方法将调用委托给 IIssueSource 实现，返回一个已经存在的问题。单元测试不应该依赖 IIssueSource 的具体实现，因此我们会使用一个 Mock。

示例 17-7：Get 方法的第一个单元测试

```
public class IssuesControllerTests
{
  private Mock<IIssueSource> _mockIssueSource = new Mock<IIssueSource>();
  private IssuesController _controller;

  public IssuesControllerTests()
  {
    _controller = new IssuesController(_mockIssueSource.Object); // <1>
  }

  [Fact]
  public void ShouldReturnIssueWhenGETForExistingIssue()
  {
    var issue = new Issue();

    _mockIssueSource.Setup(i => i.FindAsync("1"))
      .Returns(Task.FromResult(issue)); // <2>

    var foundIssue = _controller.Get("1").Result; // <3>

    Assert.Equal(issue, foundIssue); // <4>
  }
}
```

示例 17-7 主要验证的是，控制器调用 IIssueSource 实例的 FindAsync 方法，返回一个存在的问题。在测试初始化部分，代码首先使用 IIssueSource 的一个模拟实例，初始化控制器 <1>，并对这个模拟实例进行设置，使其收到参数"1"时返回一个问题 <2>，这个参数与传递给控制器的 Get 方法的参数相同 <3>。最后，控制器返回的问题与模拟实例中注入的问题相比较，验证二者相同 <4>。

这是当 IIssueSource 实现返回一个问题时的情况，但是我们也希望测试当所请求的问题没找到时，控制器如何做出反应。我们将创建一个新的测试，验证这一场景（参见示例 17-8）。

示例 17-8：覆盖问题不存在情况的单元测试

```
[Fact]
public void ShouldReturnNotFoundWhenGETForNonExistingIssue()
{
  _mockIssueSource.Setup(i => i.FindAsync("1"))
    .Returns(Task.FromResult((Issue)null)); // <1>

  var ex = Assert.Throws<AggregateException>(() =>
  {
    var task = _controller.Get("1");
    var result = task.Result;
  }); // <2>

  Assert.IsType<HttpResponseException>(ex.InnerException); // <3>
  Assert.Equal(HttpStatusCode.NotFound,
    ((HttpResponseException) ex.InnerException).Response.StatusCode); // <4>
}
```

当所请求的问题不存在时，控制器会抛出一个状态码为 404 的 HttpException。单元测试对 Mock 进行初始化，使其在问题 ID 为 1 时返回一个结果为 null 的 Task。在单元测试的断言部分，我们使用 Assert 类的 Throws 方法（由 xUnit.NET 提供），检验方法是否返回一个异常 <2>。这个 Throws 方法使用一个可能抛出异常的委托为参数，并尝试捕获这个异常。最后，我们验证抛出的异常类型是否为 HttpResponseException <3>，异常的状态码是否为 404 <4>。

2. 测试Post方法

控制器中的 Post 方法创建一个新问题。我们需要验证这个问题正确传递给了 IIssueSource 实现，并且响应在离开控制前之前正确设置了标头。这个控制器模型控制器上下文中的依赖 HttpRequestMessage 和 UrlHelper，生成响应和指向新资源的链接，因此我们需要进行一些枯燥的工作，初始化运行时配置和路由表（参见示例 17-9）。

示例 17-9：初始化运行时配置和路由表

```
_controller.Configuration = new HttpConfiguration(); // <1>

var route = _controller.Configuration.Routes.MapHttpRoute(
  name: "DefaultApi",
  routeTemplate: "api/{controller}/{id}",
  defaults: new { id = RouteParameter.Optional }
); // <2>

var routeData = new HttpRouteData(route,
  new HttpRouteValueDictionary
  {
    { "controller", "Issues" }
  }
);

_controller.Request = new HttpRequestMessage(HttpMethod.Post,
   "http://test.com/issues"); // <3>
_controller.Request.Properties.Add(HttpPropertyKeys.HttpConfigurationKey,
controller.Configuration);
_controller.Request.Properties.Add(HttpPropertyKeys.HttpRouteDataKey,
  routeData); // <4>
```

这段代码中在控制器实例中设置了一个新初始化的 HttpConfiguration 对象 <1>，设置了一个路由 <2> 并添加到已有的配置对象中，并创建了一个新的请求对象，为其设置 HTTP 动词和测试预期的 URI <3>。最后，代码通过通用属性 Properties，将路由数据和配置对象关联到请求对象，UrlHelper 会使用 Properites 查找这些对象 <4>。

我们在下一节中将看到，ASP.NET Web API 团队已经在 Web API 中简化了这一场景。同时，如果你仍然在使用 Web API 的第一个版本，WebApiContrib 项目（参见 http://www.nuget.org/packages/WebApiContrib.Testing）也提供一组扩展方法，可以用一行代码完成对控制器的配置（参见示例 17-10）。

示例 7-10：用于在测试中配置 ApiController 的扩展方法

```
public static class ApiControllerExtensions
{
  public static void ConfigureForTesting(this ApiController controller,
    HttpRequestMessage request,
    string routeName = null,
    IHttpRoute route = null);

  public static void ConfigureForTesting(this ApiController controller,
    HttpMethod method,
    string uri,
    string routeName = null,
    IHttpRoute route = null);
}
```

这些扩展方法的参数包括要使用的请求实例或者 HTTP 方法，还可以传入在测试中使用的 URL 或默认路由。我们将在本章中使用这些扩展方法，以简化测试代码。示例 17-11 列出了我们的第一个单元测试。

示例 17-11：第一个测试，验证调用 CreateAsync

```
[Fact]
public void ShouldCallCreateAsyncWhenPOSTForNewIssue()
{
  // 准备
  _controller.ConfigureForTesting(HttpMethod.Post, "http://test.com/issues"); // <1>
  var issue = new Issue();
  _mockIssueSource.Setup(i => i.CreateAsync(issue))
    .Returns(() => Task.FromResult(issue)); // <2>

  // 操作
  var response = _controller.Post(issue).Result; // <3>

  // 断言
  _mockIssueSource.Verify(i=>i
    .CreateAsync(It.Is<Issue>(iss => iss.Equals(issue)))); // <4>
}
```

测试使用扩展方法 `ConfigureForTesting`，对 `IssuesController` 中的 `HttpRequestMessage` 和 `UrlHelper` 进行实例化 <1>。控制器初始化完成之后，测试将 `IIssueSource` 模拟实例设置为返回一个异步任务，模拟后台的持久化操作 <2>，并使用一个新问题调用控制器的 `Post` 方法 <3>。测试验证模拟实例的 `CreateAsync` 方法确实得到了调用 <4>。

我们还需要一个测试，验证调用 `IIssueSource` 实现的 `CreateAsync` 方法后，返回了一个有效的响应。具体实现请见示例 17-12.

示例 17-12：第二个测试，验证响应消息

```
[Fact]
public void ShouldSetResponseHeadersWhenPOSTForNewIssue()
{
```

```
    // 准备
    _controller.ConfigureForTesting(HttpMethod.Post, "http://test.com/issues");
    var createdIssue = new Issue();
    createdIssue.Id = "1";
    _mockIssueSource.Setup(i => i.CreateAsync(createdIssue)).Returns(() =>
    Task.FromResult(createdIssue)); // <1>

    // 操作
    var response = _controller.Post(createdIssue).Result; // <2>

    // 断言
    response.StatusCode.ShouldEqual(HttpStatusCode.Created); // <3>
    response.Headers.Location.AbsoluteUri.ShouldEqual("http://test.com/issues/1");
  }
```

这个测试对 `IIssueSource` 模拟对象进行设置，使其返回结果为已创建问题的任务 <1>，这个问题作为参数传递给控制器示例的 `Post` 方法 <2>。测试验证返回的 HTTP 状态码等于 `Created` <3>，而且新资源的地址为 `http://test.com/issues/1`。到这里，你应该对 ASP.NET Web API 中控制器的单元测试有了一定的了解。在接下来的小节中，我们将讨论如何测试 `MediaTypeFormatter` 和 `HttpMessageHandler`。

3. Web API 2中的IHttpActionResult

Web API 2 中引入了一个新接口 `IHttpActionResult`（等同于 ASP.NET MVC 中的 `ActionResult`），极大简化了对控制器的单元测试。现在，一个控制器方法可以返回 `IHttpActionResult` 的一个实现，其内部使用 `Request` 或 `UrlHelper` 生成链接，因此单元测试可以只关注返回的 `IHttpActionResult` 实例。下面的代码展示了使用 `IHttpActionResult` 实例的 `Post` 方法：

```
public async Task<IHttpActionResult> Post(Issue issue)
{
  var createdIssue = await _issueSource.CreateAsync(issue);
  var result = new CreatedAtRouteNegotiatedContentResult<Issue>(
    "DefaultApi",
    new Dictionary<string, object> { { "id", createdIssue.Id } },
      createdIssue,
      this);

  return result;
}
```

`CreatedAtRouteNegotiatedContentResult` 也是由框架提供的一个实现，创建了一个新资源，设置响应消息中的资源地址。单元测试也得到了很大的简化，如示例 17-13 所示。

示例 17-13：第二个测试，验证响应消息

```
[Fact]
public void ShouldSetResponseHeadersWhenPOSTForNewIssue()
{
  // 准备
```

```
    var createdIssue = new Issue();
    createdIssue.Id = "1";
    _mockIssueSource.Setup(i => i.CreateAsync(createdIssue)).Returns(() =>
    Task.FromResult(createdIssue));

    // 操作
    var result = _controller.Post(createdIssue).Result as
    CreatedAtRouteNegotiatedContentResult; // <1>

    // 断言
    result.ShouldNotBeNull(); // <2>
    result.Content.ShouldBeType<Issue>(); // <3>
  }
```

这个单元测试只是返回的结果转换为预期的类型（在这个测试中是 `CreatedAtRouteNegotiatedContentResult`）<1>，并验证结果不为 `null` <2>，而且结果中的内容为 `Issue` 类型的一个实例 <3>。这个测试不再需要之前使用的初始化代码，因为现在所有的内容协商和链接管理逻辑都封装在 `IHttpActionResult` 实现之中，和这个单元测试无关。

17.2.2　测试MediaTypeFormmater

作为处理新媒体类型或内容协商的主要部分，`MediaTypeFormatter` 的实现有几个方面需要在单元测试中解决。这些方面有：正确处理所支持的媒体类型，将模型从指定媒体类型进行转换或者转换到指定媒体类型，或者在需要时检查某些设置的正确配置，例如：编码或者映射。

从示例 17-14 中的 `MediaTypeFormatter` 类定义，你可以对其单元测试得到一个粗略的概念。

示例 17-14：`MediaTypeFormatter` 类定义

```
public abstract class MediaTypeFormatter
{
  public Collection<Encoding> SupportedEncodings { get; }

  public Collection<MediaTypeHeaderValue> SupportedMediaTypes { get; }

  public Collection<MediaTypeMapping> MediaTypeMappings { get; }

  public abstract bool CanReadType(Type type);

  public abstract bool CanWriteType(Type type);

  public virtual Task<object> ReadFromStreamAsync(Type type, Stream readStream,
    HttpContent content, IFormatterLogger formatterLogger);

  public virtual Task WriteToStreamAsync(Type type, object value,
    Stream writeStream, HttpContent content, TransportContext transportContext);
}
```

使用不同的单元测试，我们可以进行以下检验。

- 所支持的媒体类型（参见示例 17-15）在 SupportedMediaType 集合中进行了正确配置。对于我们在第 13 章中构建的支持联合媒体类型（如 Atom 或 RSS）的格式化程序，这个测试意味着对于所支持的这些媒体类型，集合中分别包含 application/atom+xml 和 application/rss+xml。

示例 17-15：检查所支持媒体类型的单元测试

```
[Fact]
public void ShouldSupportAtom()
{
  var formatter = new SyndicationMediaTypeFormatter();

  Assert.True(formatter.SupportedMediaTypes
    .Any(s => s.MediaType == "application/atom+xml"));
}

[Fact]
public void ShouldSupportRss()
{
  var formatter = new SyndicationMediaTypeFormatter();

  Assert.True(formatter.SupportedMediaTypes
    .Any(s => s.MediaType == "application/rss+xml"));
}
```

- 在 CanReadType 和 CanWriteType 方法中，支持给定模型类型的序列化或反序列化（参见示例 17-16）。

示例 17-16：检查实现是否能够读或写一个类型的单元测试

```
[Fact]
public void ShouldNotReadAnyType()
{
  var formatter = new SyndicationMediaTypeFormatter();

  var canRead = formatter.CanReadType(typeof(object));

  Assert.False(canRead);
}

[Fact]
public void ShouldWriteAnyType()
{
  var formatter = new SyndicationMediaTypeFormatter();

  var canWrite = formatter.CanWriteType(typeof(object));

  Assert.True(canWrite);
}
```

- 使用所支持的媒体类型，从流读取模型 / 向流写入模型的代码正常工作（参见示例 17-17）。这需要分别测试 WriteToStreamAsync 和 ReadFromStreamAsync 方法。

示例 17-17：验证 `WriteToStreamAsync` 方法行为的单元测试

```
[Fact]
public void ShouldSerializeAsAtom()
{
  var ms = new MemoryStream();

  var content = new FakeContent();
  content.Headers.ContentType = new MediaTypeHeaderValue("application/atom+xml");

  var formatter = new SyndicationMediaTypeFormatter();

  var task = formatter.WriteToStreamAsync(typeof(List<ItemToSerialize>),
    new List<ItemToSerialize> { new ItemToSerialize { ItemName = "Test" }},
    ms,
    content,
    new FakeTransport()
  );

  task.Wait();

  ms.Seek(0, SeekOrigin.Begin);

  var atomFormatter = new Atom10FeedFormatter();
  atomFormatter.ReadFrom(XmlReader.Create(ms));

  Assert.Equal(1, atomFormatter.Feed.Items.Count());
}

public class ItemToSerialize
{
  public string ItemName { get; set; }
}

public class FakeContent : HttpContent
{
  public FakeContent()
    : base()
  {
  }

  protected override Task SerializeToStreamAsync(Stream stream, TransportContext
    context)
  {
    throw new NotImplementedException();
  }

  protected override bool TryComputeLength(out long length)
  {
    throw new NotImplementedException();
  }
}

public class FakeTransport : TransportContext
{
  public override ChannelBinding GetChannelBinding(ChannelBindingKind kind)
```

```
    {
      throw new NotImplementedException();
    }
  }
```

示例 17-17 中的单元测试序列化了一组 ItemToSerialize 类型的条目，这组条目在测试中也定义为一个 Atom 源。这个测试主要验证了，当内容类型为 application/atom+xml 时，SyndicationMediaTypeFormatter 能够序列化这组条目。WriteToStreamAsync 方法需要 HttpContent 和 TransportContext 实例，而这两个实例在整个实现中并没有使用（标头除外），因此测试定义了两个不包含任何特殊逻辑的伪造类。测试还使用 WCF 的联合类，将流反序列化为一个 Atom 源，以确保序列化操作正确完成。

- 所有需要的设置都正确进行了初始化。例如，如果你要求在查询字符串中支持媒体类型映射，那么单元测试可以检查这个映射是否在 MediaTypeMappings 集合中正确进行了配置。示例 17-18 对此进行了演示。

示例 17-18：检查所支持媒体类型映射的单元测试

```
[Fact]
public void ShouldMapAtomFormatInQueryString()
{
  var formatter = new SyndicationMediaTypeFormatter();

  Assert.True(formatter.MediaTypeMappings.OfType<QueryStringMapping>()
    .Any(m => m.QueryStringParameterName == "format" &&
              m.QueryStringParameterValue == "atom" &&
              m.MediaType.MediaType == "application/atom+xml"));
}
```

示例 17-18 中的测试检查是否为查询字符串参数定义了一个 MediaTypeMapping，将值为 atom 的查询字符串变量 format 映射到媒体类型 application/atom+xml。

17.2.3　单元测试HttpMessageHandler

HttpMessageHandler 是 Web API 运行时管道的通用拦截机制。消息处理程序是异步的，通常只包含一个方法 SendAsync，用于处理请求消息（HttpRequestMessage），并返回一个 Task 实例，代表获取一个响应（HttpResponseMessage）的任务（参见示例 17-19）。

示例 17-19：HttpMessageHandler 类定义

```
public abstract class HttpMessageHandler
{
  protected internal abstract Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request, CancellationToken cancellationToken);
}
```

SendAsync 方法不是公开的，因此无法在单元测试中直接调用，但是框架提供了一个 System.Net.Http.MessageInvoker 类，可以用于调用 SendAsync 方法。System.Net.Http.

MessageInvoker 类的构造函数参数是一个 HttpMessageHandler 实例，并提供一个公共方法 SendAsync，用于调用这个处理程序上的同名方法。示例 17-20 简单展示了如何对一个 HttpMessageHandler 示例的 SendAsync 方法进行单元测试。但是，HttpMessageHandler 可能使用外部依赖或者包含一些其他的公共方法，你也需要对其进行测试。

示例 17-20：对 HttpMessageHandler 进行单元测试

```
[Fact]
public void ShouldInvokeHandler()
{
    var handler = new SampleHttpMessageHandler();

    var invoker = new HttpMessageInvoker(handler);
    var task = invoker.SendAsync(new HttpRequestMessage(), new CancellationToken());
    task.Wait();

    var response = task.Result;

    // 对响应进行断言
    // ......
}
```

17.2.4 测试ActionFilterAttribute

操作筛选器和 HTTP 消息处理程序一样都是消息拦截机制，但是在操作上下文初始化之后，操作即将执行之前在运行时管道中运行得更深入。操作筛选器的基类 System.Web.Http.Filters.ActionFilterAttribute（参见示例 17-21）提供两个可以重写的方法：OnActionExecuting 和 OnActionExcuted，可以在操作执行前后拦截调用。

示例 17-21：ActionFilterAttribute 类定义

```
public abstract class ActionFilterAttribute : FilterAttribute,
  IActionFilter,
  IFilter
{
  public virtual void OnActionExecuted(HttpActionExecutedContext
    actionExecutedContext);

  public virtual void OnActionExecuting(HttpActionContext
    actionContext);
}
```

这两个方法都是公开的，因此可以直接在单元测试中调用。示例 17-22 是一个非常基础的过滤器实现，使用应用程序码对客户端进行身份验证。我们将使用这个具体实现，来展示如何对不同的场景进行单元测试。

示例 17-22：使用应用程序码对客户端进行身份验证的操作筛选器

```
public interface IKeyVerifier
{
  bool VerifyKey(string key);
```

```
}

public class ApplicationKeyActionFilter : ActionFilterAttribute
{
  public const string KeyHeaderName = "X-AuthKey";

  IKeyVerifier keyVerifier;

  public ApplicationKeyActionFilter()
  {
  }

  public ApplicationKeyActionFilter(IKeyVerifier keyVerifier) // <1>
  {
    this.keyVerifier = keyVerifier;
  }

  public Type KeyVerifierType // <2>
  {
    get;
    set;
  }

public override void OnActionExecuting(HttpActionContext
  actionContext)
{
    if (this.keyVerifier == null)
    {
        if (this.KeyVerifierType == null)
        {
          throw new Exception("The keyVerifierType was not provided");
        }

        this.keyVerifier = (IKeyVerifier)Activator
          .CreateInstance(this.KeyVerifierType);
    }

    IEnumerable<string> values = null;

    if (actionContext.Request.Headers
      .TryGetValues(KeyHeaderName, out values)) // <3>
    {
        var key = values.First();

        if (!this.keyVerifier.VerifyKey(key)) // <4>
        {
            actionContext.Response =
                new HttpResponseMessage(HttpStatusCode.Unauthorized);
        }
    }
    else
    {
        actionContext.Response =
          new HttpResponseMessage(HttpStatusCode.Unauthorized);
    }
```

```
    base.OnActionExecuting(actionContext);
  }
}
```

示例中的这个操作筛选器的构造函数参数为一个 IKeyVerifier 实例，用于验证一个码值是否有效 <1>。操作筛选器也可以用做属性，因此这个实现提供一个属性 KeyVerifierType <2>，在筛选器用做属性时设置 IKeyVerifier。这个筛选器只实现了 OnActionExecuting 方法，这个方法在操作执行之前运行。这个筛选器检查上下文中设置的请求消息的 X-Auth 标头 <3>，并尝试将这个标头值传给 IKeyVerifier 实例进行验证 <4>。如果这个码值无法得到验证或者没有在请求消息中找到，那么这个筛选器会在当前上下文中设置一个 HTTP 状态码为 401 的响应消息，管道执行因此中断。

示例 17-23 中的第一个单元，测试请求消息中传入有效的码值的场景。

示例 17-23：有效码值的单元测试

```
[Fact]
public class ApplicationKeyActionFilterFixture
{
  public void ShouldValidateKey()
  {
      var keyVerifier = new Mock<IKeyVerifier>();
      keyVerifier
          .Setup(k => k.VerifyKey("mykey"))
          .Returns(true); // <1>

      var request = new HttpRequestMessage();
      request.Headers.Add("X-AuthKey", "mykey"); // <2>

      var actionContext = InitializeActionContext(request); // <3>

      var filter = new ApplicationKeyActionFilter(keyVerifier.Object);
      filter.OnActionExecuting(actionContext); // <4>

      Assert.Null(actionContext.Response); // <5>
  }
}

private HttpActionContext InitializeActionContext(HttpRequestMessage request)
{
    var configuration = new HttpConfiguration();

    var route = configuration.Routes.MapHttpRoute(
      name: "DefaultApi",
      routeTemplate: "api/{controller}/{id}",
      defaults: new { id = RouteParameter.Optional }
    );

    var routeData = new HttpRouteData(route,
        new HttpRouteValueDictionary
        {
```

```
                { "controller", "Issues" }
            }
        );

        request.Properties[HttpPropertyKeys.HttpRouteDataKey] = routeData;

        var controllerContext = new HttpControllerContext(configuration, routeData,
    request);

        var actionContext = new HttpActionContext {
            ControllerContext = controllerContext
        };

        return actionContext;
    }
```

这个单元测试的第一步 <1>，是初始化 IKeyVerfier 的一个模拟或伪实例，这个实例的 VerifyKey 方法在传入参数为 mykey 时返回 true。第二步 <2>，测试创建了一个新的 HTTP 请求消息，将定制标头 X-AuthKey 的值设置为 IKeyVerifier 实例预期的码值。方法 InitializeActionContext 初始化了筛选器预期的操作上下文 <3>，其中用到了大量的通用基础代码，将路由配置和请求消息注入到 HttpControllerContext 类的构造函数中。最后，测试使用完成初始化的上下文，调用 OnActionExecuting 方法 <4>，并对 null 响应进行了断言 <5>。如果这个操作筛选器实现中没有出现错误，那么上下文中不会设置响应，这个测试就会通过。

示例 17-24 将测试下一个场景，即码值无效并返回一个状态码为 401 (Unauthorized) 的响应。

示例 17-24：无效码值的单元测试

```
    [Fact]
    public void ShouldNotValidateKey()
    {
        var keyVerifier = new Mock<IKeyVerifier>();
        keyVerifier
            .Setup(k => k.VerifyKey("mykey"))
            .Returns(true);

        var request = new HttpRequestMessage();
        request.Headers.Add(ApplicationKeyActionFilter.KeyHeaderName, "badkey"); // <1>

        var actionContext = InitializeActionContext(request);

        var filter = new ApplicationKeyActionFilter(keyVerifier.Object);
        filter.OnActionExecuting(actionContext);

        Assert.NotNull(actionContext.Response); // <2>
        Assert.Equal(HttpStatusCode.Unauthorized,
            actionContext.Response.StatusCode); // <3>
    }
```

这个测试与上一个测试的主要区别在于，请求消息中设置的应用程序码 <1> 与

IKeyVerifer 模拟实例预期的码值不同。在调用 OnActionExecuting 方法之后，测试执行了两个断言，以确保上下文中设置的响应不为 null <2>，而且响应的状态码为 401 (Unauthorized) <3>。

17.3 对路由进行单元测试

路由配置也是你可能需要在单元测试中覆盖的一个方面。虽然路由配置本身不是一个组件，但是一个不遵循常用规范的复合路由配置可能会导致一些问题，这些问题需要尽早发现，在实现部署之前解决。不幸的是，ASP.NET Web API 没有直接提供对路由单元测试的任何支持，因此我们需要使用定制代码。定制代码通常会使用 Web API 基础结构中内建的一些组件，例如，DefaultHttpControllerSelector 和 ApiControllerActionSelector，为给定的 HttpRequestMessage 和路由配置推导出控制器类型和操作名。具体实现请见示例 17-25。

示例 17-25：测试路由的一个泛型方法

```
public static class RouteTester
{
  public static void TestRoutes(HttpConfiguration configuration,
    HttpRequestMessage request,
    Action<Type, string> callback)
  {
    var routeData = configuration.Routes.GetRouteData(request);
    request.Properties[HttpPropertyKeys.HttpRouteDataKey] = routeData;

    var controllerSelector = new DefaultHttpControllerSelector(configuration);// <1>
    var controllerContext = new HttpControllerContext(configuration, routeData,
      request);

    controllerContext.ControllerDescriptor = controllerSelector
      .SelectController(request); // <2>

    var actionSelector = new ApiControllerActionSelector(); // <3>

    var action = actionSelector.SelectAction(controllerContext).ActionName; // <4>
    var controllerType = controllerContext.ControllerDescriptor
      .ControllerType; // <5>

    callback(controllerType, action); // <5>
  }
}
```

示例 17-25 实现了一个泛型方法。这个方法的参数是带有路由配置的 HttpConfiguration 实例和一个 HttpRequestMessage，使用所选的控制器类型和操作名调用一个回调函数。这个方法首先使用参数中传入的 HttpConfiguration，实例化一个 DefaultHttp ControllerSelector，用于决定控制器类型 <1>；然后选中这个控制器，并将 HttpRequestMessage 作为参数传入 <2>。选中控制器之后，这个方法接着实例化一个 ApiControllerActionSelector，用于获取操作名 <3>，并在 <4> 和 <5> 中获得操作名和控制器类型。最后，这个方法使用获得的

控制器类型和操作名，调用一个回调函数 <5>。单元测试可以使用这个回调函数执行断言。具体用法请见示例 17-26.

示例 17-26：使用 `RouteTester` 实现的单元测试

```
[Fact]
public void ShouldRouteToIssueGET()
{
  var config = new HttpConfiguration();
  config.Routes.MapHttpRoute(name: "Default",
    routeTemplate: "api/{controller}/{id}"); // <1>

  var request = new HttpRequestMessage(HttpMethod.Get,
  "http://www.example.com/api/Issues/1"); // <2>

  RouteTester.TestRoutes(config, request, // <3>
    (controllerType, action) =>
    {
      Assert.Equal(typeof(IssuesController), controllerType);
      Assert.Equal("Get", action);
    });
}
```

示例 17-26 展示了如何在单元测试中使用 `RouteTester` 类，验证路由配置。这个测试实例化了一个 `HttpConfiguration`，为其配置了需要测试的路由 <1>，以及带有 HTTP 谓词和待调用 URL 的 `HttpRequestMessage` <2>。在最后一步，测试使用 `RouteTester`，从配置和请求实例得到控制器类型和操作名。在回调函数中，测试定义了断言，将获得的控制器类型和操作名与预期值进行比较 <3>。

17.4 ASP.NET Web API的集成测试

到目前为止，我们讨论的都是单元测试，单元测试的关注点是在隔离环境下测试组件。但是，如果要测试所有组件在指定场景下的相互协作，该怎么做呢？这时你需要用到集成测试。对于 Web API，集成测试更为关注的是，测试从客户端到服务的一个完整的端到端调用，其中包括栈中的所有组件，如控制器、筛选器、消息处理程序，或者你的 Web API 运行时中配置的任何其他组件。例如，你可能想使用 `HttpMessageHandler` 支持基础身份验证，需要进行集成测试，从客户端应用程序的角度，验证这个处理程序与已有的控制器的行为。理想情况下，你也会对这些组件进行单元测试，以确保组件单独运行时的行为是正确的。

在 ASP.NET Web API 中进行集成测试，我们要使用 `HttpClient`，这个类可以在内存托管服务器中处理请求。使用这种方法，我们无需打开端口或通过网络发送消息，对于简化测试具有明显的优点。如示例 17-27 所示，`HttpClient` 类定义了几个以 `HttpMessageHandler` 实例为参数的构造函数。我们在第 4 章中介绍过，`HttpServer` 类是 `HttpMessageHandler` 的一个实现，因此可以直接注入 `HttpClient` 实例中，自动处理客户端在测试中发送的任何

消息。

示例 17-27：HttpClient 构造函数

```
public class HttpClient : HttpMessageInvoker
{
  public HttpClient();
  public HttpClient(HttpMessageHandler handler);
  public HttpClient(HttpMessageHandler handler, bool disposeHandler);
}
```

在集成测试中，我们可以使用之前实现的 HttpMessageHandler，对一个 HttpServer 实例进行配置，并将这个 HttpServer 传给 HttpClient，对端到端的测试场景进行验证。具体实现请见示例 17-28。

示例 17-28：基础身份验证的集成测试

```
public class BasicAuthenticationIntegrationTests
{
  [Fact]
  public ShouldReturn404IfCredentialsNotSpecified()
  {
      var config = new HttpConfiguration();
      config.Routes.MapHttpRoute(name: "Default",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional }); // <1>

      config.MessageHandlers.Add(new BasicAuthHttpMessageHandler()); // <2>

      var server = new HttpServer(config);

      var client = new HttpClient(server); // <3>

      var task = client.GetAsync("http://test.com/issues"); // <4>
      task.Wait();

      var response = task.Result;
      Assert.AreEqual(HttpStatusCode.Unauthorized, response.StatusCode); // <5>
  }
}
```

如示例 17-28 所示，我们仍然可以使用单元测试框架进行集成测试的自动化。我们的测试为一个内存服务器配置了默认路由 <1> 和一个 BasicAuthHttpMessageHandler <2>，这个处理程序内部实现了基础身份验证。测试将这个服务器注入到 HttpClient <3>，因此使用 GetAsync 调用 http://test.com/issues 会将请求发送到这个服务器 <4>。在这个测试中，我们没有在 HttpClient 中设置身份验证标头，因此测试预期 BasicAuthHttpMessageHandler 返回一个状态码为 404 (Unauthorized) 的响应消息 <5>。身份验证只是集成测试适用的一个场景，但是你可以想象得到，集成测试也可以扩展到任何需要多个组件协作的情况。

17.5 小结

测试驱动开发是驱动 Web API 设计和实现的极为有效的工具。测试驱动开发的副产品是反映 API 实现预期行为的单元测试。你可以使用这些单元测试，逐步对已有代码进行改进。当代码中引入新变更时，这些测试还可以用来确保已有功能没有遭到破坏，并且系统行为依然符合预期。测试驱动开发中有两种常用实践：依赖注入和代码重构。依赖注入关注的是，如何移除显式依赖，生成更容易测试的代码；代码重构则用于提高现有代码的质量。单元测试关注如何在隔离环境中测试特定代码，除此之外，你也可以使用集成测试，对端到端的场景进行测试，验证系统中的不同组件如何进行交互。

附录A

媒体类型

表A-1：媒体类型

媒体类型	描　　述	参考资料
text/html	用于交换 HTML 文档	http://www.iana.org/assignments/media-types/text/html
application/xhtml+xml	用于交换使用格式良好的 XML 的 HTML 文档	http://tools.ietf.org/html/rfc3236
application/xml	用于交换 XML 文档和模式	http://www.rfc-editor.org/rfc/rfc3023.txt
application/json	用于交换 JSON 文档	http://www.ietf.org/rfc/rfc4627.txt
application/x-www-form-urlencoded	用于交换表单键 / 值数据	
multipart/mixed	用于交换多个数据集组合而成的单个正文	http://tools.ietf.org/html/rfc1521#section-7.2.2
multipart/form-data	主要用于交换文件	http://tools.ietf.org/html/rfc2388
image/jpeg	用于交换 JPEG 文档	http://tools.ietf.org/html/rfc2046
image/gif	用于交换 GIF 文档	http://tools.ietf.org/html/rfc2046
image/png	用于交换 PNG 文档	http://tools.ietf.org/html/rfc2083
image/svg+xml	用于交换 SVG（http://www.w3.org/TR/SVG11/）文档	http://www.w3.org/TR/SVG/mimereg.html
application/atom+xml	用于交换 Atom 源	http://tools.ietf.org/html/rfc4287
application/vnd.hal+json	用于交换包含相关资源链接的数据	http://stateless.co/hal_specification.html
application/vnd.collection+json	用于管理数据集合	http://amundsen.com/media-types/collection

附录B

HTTP标头

表B-1：消息头

标　　头	描　　述	参考资料
Cache-Control	向请求 / 响应经过的缓存机制给出消息可缓存性的指令	http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.2
Connection	给出与当前链接相关的选项，不应传给代理	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-6.1
Date	说明消息产生的日期和时间	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.1.1.2
Pragma	向缓存说明应该总是重新验证已缓存的响应。这个标头是为了向后兼容 HTTP 1.0 客户端，在 HTTP 1.1 中由 Cache-Control 标头取代	http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.4
Transfer-Encoding	表明消息正文是否为了在发送方和接收方之间传输而进行了转码	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-3.3.1
Upgrade	如果服务器愿意切换的话，允许客户端指定希望使用附加协议	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-6.3
Via	由网管和代理使用，包含请求时在客户端和服务器之间，以及响应时在服务器和客户端之间的中间协议和接收者。响应 TRACE 请求的消息中的这个标头非常有用	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-5.7
Warning	用于传递可能无法包含在消息自身的附加消息信息	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-5.7

表B-2：请求标头

标　　头	描　　述	参考资料
Host	提供目标 URI 中的主机和端口信息	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-5.4
Max-Forwards	用于使用 TRACE 和 OPTION 方法进行调试，允许客户端对代理转发请求的次数进行限制	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.1.1
Expect	告知服务器客户端预期的行为。例如：`Expect: 200-Continue` 告知服务器，客户端预期在开始发送正文之前处理请求	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.1.2
Range	指明服务器应该执行一个 byte-range 操作，只返回请求的字节	http://tools.ietf.org/html/draft-ietf-httpbis-p5-range-21#section-5.4
If-Match	用于进行条件请求，只有实体的标签值匹配资源的一个或多个表示时才执行请求	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-3.1
If-None-Match	用于执行条件请求，只有实体的标签值不匹配资源的一个或多个表示时才执行请求	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-3.2
If-Modified-Since	用于执行条件请求，只有当资源在指定时间后经过修改才执行请求	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-3.3
If-Unmodified-Since	用于执行条件请求，只有当资源在指定时间后未经过修改才执行请求	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-3.4
If-Range	用于执行条件请求，只要实体标签匹配，就允许客户端获取返回部分表示	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-3.5
Accept	包含一个带优先级的列表，列出可接受的响应媒体类型	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.3.2
Accept-Charset	包含一个带优先级的列表，列出可接受的响应字符编码	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.3.3
Accept-Encoding	包含一个带优先级的列表，列出可接受的传输编码	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.3.4
Accept-language	包含一个带优先级的语言列表	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.3.5
From	指明请求发起人的电子邮件地址	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.5.1
Referer	指明为当前请求提供目标 URI 的资源的 URI	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.5.2
TE	指出除了“chunked”之外，可接受的传输编码	http://tools.ietf.org/html/draft-ietf-httpbis-p1-messaging-21#section-4.3
User-Agent	提供生成请求的客户端的信息	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-6.5.3
Authorization	包含待访问域的身份信息	http://tools.ietf.org/html/draft-ietf-httpbis-p7-auth-21#section-4.1

表B-3：响应标头

标　　头	描　　述	参考资料
Age	指明响应生成后经过的时间	http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.1
Date	指明消息生成的日期和时间	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.1.1.2
Location	指明与这个响应相关的一个资源（已创建的资源或客户端应该重定向到的资源）	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.1.1.2
Retry-After	指明客户端在向资源重试请求前应该等待的时间。在重定向响应中，这个标头与重定向 URI 相关	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.1.1.3
Last-Modified	指明源服务器认为资源受到修改的日期和时间	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-2.2
ETag	指明当前所选表示的唯一标识符	http://tools.ietf.org/html/draft-ietf-httpbis-p4-conditional-21#section-2.3
Vary	指明哪些字段用于选择向客户端返回的表示	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.2.1
WWW-Authenticate	指明一个或多个身份验证质询，告知客户端必须如何向目标资源提供身份验证	http://tools.ietf.org/html/draft-ietf-httpbis-p7-auth-21#section-4.4
Proxy-Authenticate	指明一个或多个身份验证质询，告知客户端必须如何向目标资源的代理提供身份验证	http://tools.ietf.org/html/draft-ietf-httpbis-p7-auth-21#section-4.2
Accept-Ranges	指明客户端在使用范围请求时能使用的可接受范围	http://tools.ietf.org/html/draft-ietf-httpbis-p5-range-21#section-5.1
Allow	指明目标资源支持哪些 HTTP 方法	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.4.1
Server	包含源服务器的服务器环境信息	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-8.4.2

表B-4：表示标头

标　　头	描　　述	参考资料
Content-Type	指明表示的媒体类型	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-3.1.1.5
Content-Encoding	指明应用于表示的内容编码	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-3.1.2.2
Content-Language	指明当前表示的目标受众语言	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-3.1.1.5
Content-Location	指明专门用于获取当前表示的 URI	http://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-21#section-3.1.4.2
Expires	给出响应过期的日期和时间	http://tools.ietf.org/html/draft-ietf-httpbis-p6-cache-21#section-7.3

附录C
内容协商

内容协商（conneg）有两种类型：主动式（proactive）内容协商和被动式（reactive）内容协商。

主动式内容协商

如果服务器负责选择资源表示，包含寻找最佳资源表示的逻辑，在每次请求时执行，那么内容协商就是主动式的。服务器将可用的表示与客户端偏好或附加标头进行匹配，据此做出选择。客户端通过之前介绍的 `Accept*` 请求标头（参见表 B-2）表达自己的偏好。这些标头都可以发送多个值或范围，以及包含优先级的*限定符*（qualifier，也称为 q-value）。但服务器可以使用附加字段，如 `User-Agent` 或其他字段。

如果服务器认为，客户端发送的信息不够，无法据此做出选择，那么可以使用默认选择，返回一个状态码 `406 Not Acceptable`，或者执行*被动式协商*（参见下一节）。一旦做出选择，服务器应该向客户端返回所选的表示。服务器返回的响应应当包含一个 `Vary` 标头，确切指明使用了哪个标头字段做出选择。服务器还可以提供一个 `Content-Location` 标头，给出协商所得内容的 URI。需要记住的是，服务器不受客户端偏好的制约，但是应该尽量尝试满足客户端偏好。

图 C-1 展示了主动式内容协商（proactive negotiation）的过程。

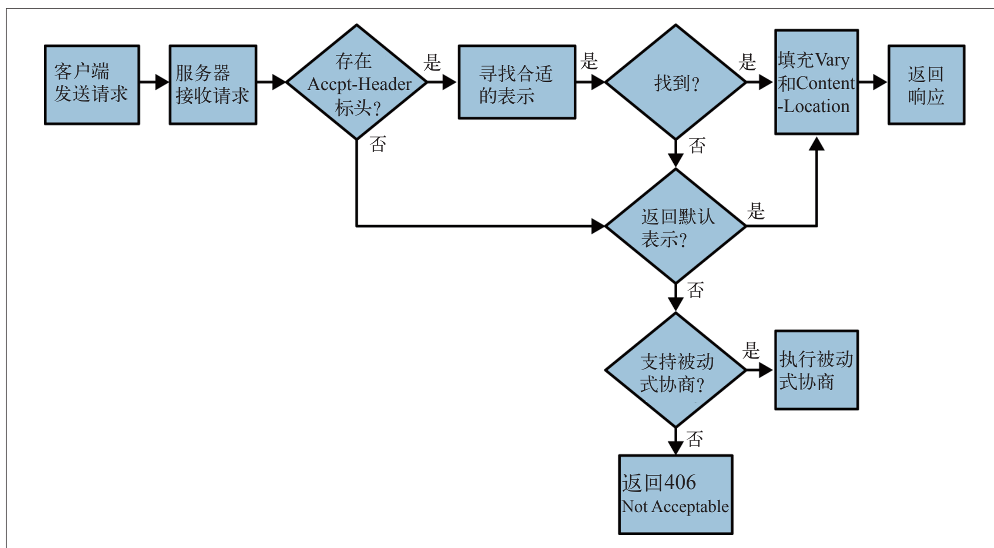


图 C-1：主动式内容协商

请注意，在图 C-1 中，如果客户端没有发送偏好信息，或者服务器没有找到合适的匹配表示，服务器可以自行做出选择，返回一个默认表示，或者返回一个 `406 Not Acceptable` 响应。

Web 浏览器通常使用主动式内容协商。当你向服务器发起请求时，浏览器会发送自己支持的一列偏好信息。在一些情况下，浏览器可能会发送浏览器插件支持的附加媒体类型。下面是使用 Chrome 浏览器发送的一个请求，请注意其中由浏览器发送的各种 `Accept` 标头。不同的浏览器也会有不同的偏好。

```
GET http://www.yahoo.com/ HTTP/1.1
Host: www.yahoo.com
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

被动式内容协商

使用被动式内容协商（reactive negotiation，也称为代理驱动协商，即 agent-driven negotiation），选择权就转到客户端一方。当客户端向服务器发送一个资源请求时，服务器会返回一个状态码为 `300 Multiple Choices` 的响应，其中包含一列表示。客户端根据自己的逻辑，从这个列表中进行选择。

图 C-2 描述了被动式内容协商的流程。

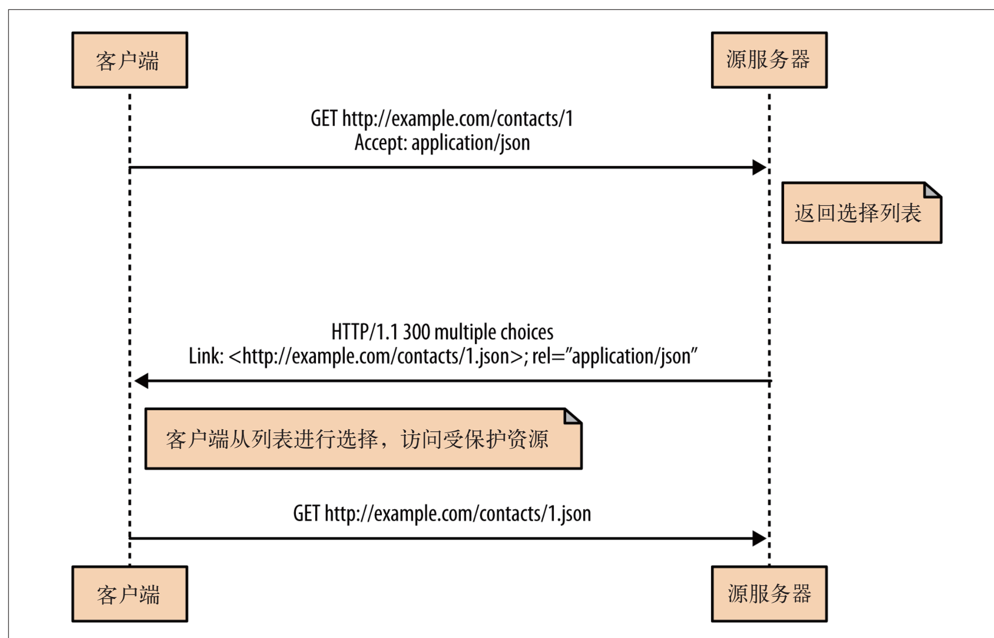


图 C-2：被动式内容协商

至于包含选择的表示自身，HTTP 协议的规范并没有进行详细规定。Mike Amundsen 撰写了一篇很好的文章"Agent-Driven conneg in HTTP"（http://www.amundsen.com/blog/archives/1085），介绍被动式内容协商。

在这篇文章中，Amundsen 推荐了几种完全可行的不同实现方法，一种是返回一个 XHTML 表示，在其中为每个选项包含一条 `<a hrefs>`，示例如下：

```
HTTP/1.1 300 Multiple Choices
Host: www.example.org
Content-Type: application/xhtml
Content-Length:XXX

<p>
  Select one:
</p>
<a href="/results/fr" hreflang="fr">French</a>
<a href="/results/en-US" hreflang="en-US">US English</a>
<a href="/results/de" hreflang="de">German</a>
```

另一个方法是使用 `Link` 标头。这种方法的优点是：`Link` 是标准标头，任何客户端都能理解。示例如下：

```
HTTP/1.1 300 Multiple Choices
Host: www.example.org
Content-Length: 0
```

```
Link: <http://www.example.org/results/png>; type="image/png",
      <http://www.example.org/results/jpeg>;type="image/jpeg",
      <http://www.example.org/results/gif>;type="image/gif"
```

使用已有机制的好处是，你可以预期任何 HTTP 客户端都理解这种机制。你可以返回 `application/json` 格式的内容，在其中嵌入链接，但是除非客户端以离线方式获得了这一信息，就无法知道如何解析你返回的结果。对此，你可以使用 profile 链接标头，指向一个定义这种链接格式的文档（无需创建新媒体类型），帮助客户端正确解析结果。在下面的示例中，profile 文档将说明 , 应使用 JSON 的 Alternate 列表解析响应内容：

```
HTTP/1.1 300 Multiple Choices
Host: www.example.org
Content-Type: application/json
Content-Length: XXX
Link: <http://www.example.org/profile>; rel="profile"

{
  "alternates" : [
    {"href": "http://www.example.org/results/png", "type":"image/png"},
    {"href": "http://www.example.org/results/jpeg", "type": "image/jpeg"},
    {"href", "http://www.example.org/results/gif", "type": "image/gif"}
  ]
}
```

附录D

缓存实战

我们已经了解到，HTTP 缓存涉及几个部分。为了说明各部分如何协同工作，我们来讨论一个常见的场景，这个场景中有两个客户端、一个 HTTP 缓存以及源服务器。为简单表示，图中的响应将省略正文和标头。

首先，如图 D-1 所示，客户端 A 发出一个初始请求。

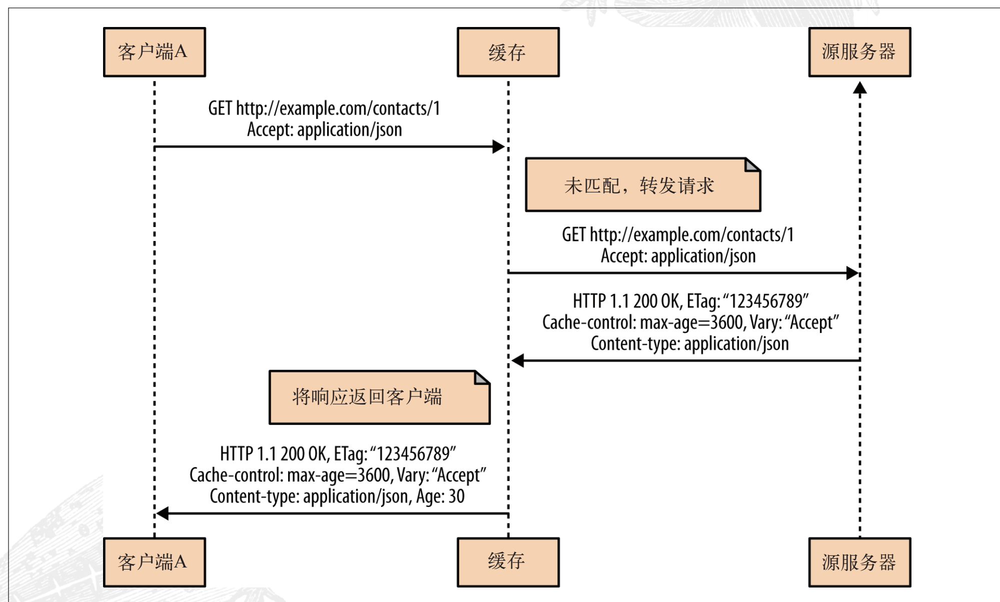


图 D-1：客户端 A 发出初始 GET 请求

(1) 缓存收到请求，看到这是一个 GET 请求，检查是否有已缓存的响应。缓存未找到匹配的响应，因此将请求向源服务器转发。
(2) 源服务器生成响应，响应中包含 ETag 和 max-age 标头。
(3) 缓存收到响应，使用请求 URI 的散列值和 Accept 标头值，将响应保存在缓存中。
(4) 缓存随后将响应返回，响应中包含一个附加的 AGE 标头，告诉客户端这个表示的存在时间。
(5) 客户端 A 收到资源表示，保存其 ETag 和 Expires 信息。

15 分钟之后，如图 D-2 所示，客户端 B 向同一个资源发起请求。

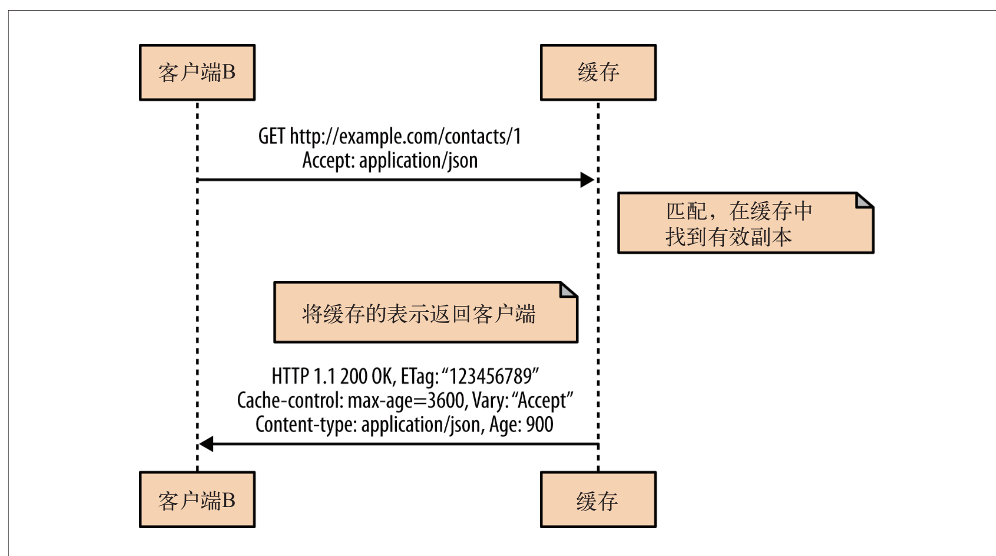


图 D-2：客户端 B 发出初始 GET 请求

(1) 缓存收到请求，检查是否有表示的副本。
(2) 缓存使用 URI 和 Accept 进行匹配，发现表示存在而且有效（根据有效期判断），因此立即返回更新了存在时间的表示。
(3) 客户端 B 收到资源表示，保存其 ETag 和 Expires 信息。

一个小时之后，如图 D-3 所示，客户端 A 又向同一资源发出一个条件 GET 请求，请求中包含了 If-None-Match 标头。

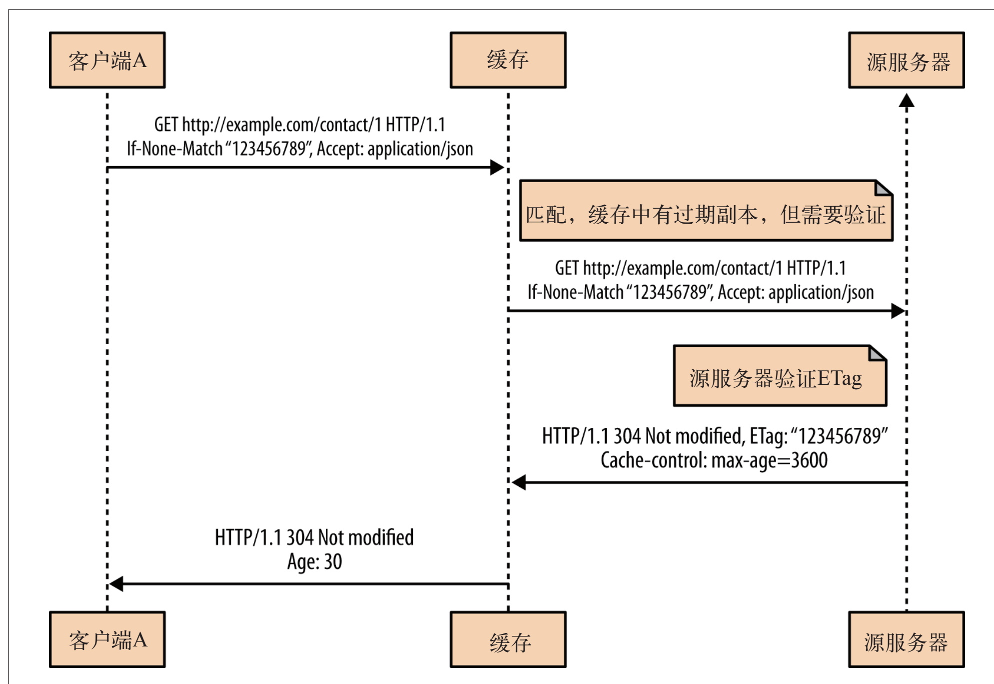


图 D-3：客户端 A 发出条件 GET 请求

(1) 缓存收到请求，检查是否存在表示副本。缓存找到匹配的表示，发现表示已经过期，随后将条件 `GET` 请求转发给源服务器，以判断缓存保存的副本是否依然有效。
(2) 源服务器收到请求，断定 `ETAG` 依然有效，返回一个状态为 `304 Not Modified` 的响应，其中包含新的 `max-age`。
(3) 缓存收到响应，向客户端返回这个 `304` 响应，并在其中包含更新的表示存在时间。

时间流逝，如图 D-4 所示，客户端 B 向联系人资源发出一个条件 `PUT` 请求，更新联系人状态。

(1) 缓存收到请求，发现这个是一个 `PUT` 请求，检查缓存是否保存了这个资源的副本，以及 `ETag` 是否匹配。找到副本后，缓存将这个 `ETag` 标记为无效，以处理将来的请求。然后缓存将请求原样转发给源服务器。
(2) 服务器执行更新操作，生成带有更新的 `ETag` 的新响应。
(3) 缓存收到响应，保存副本，然后将响应返回给客户端 B。

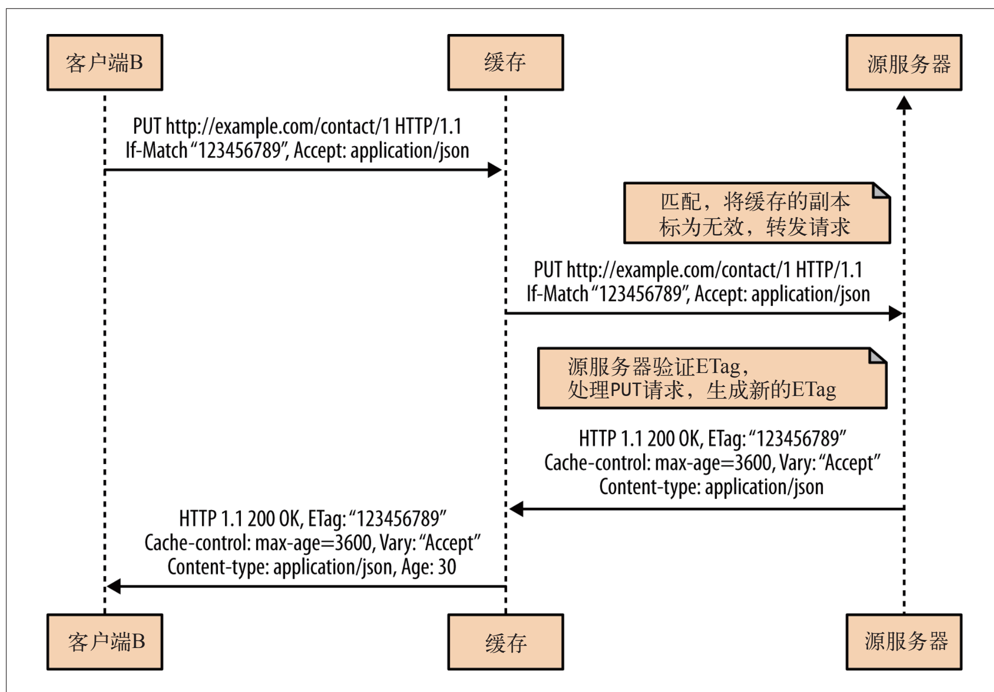


图 D-4：客户端 B 发出条件 PUT 请求

10 分钟之后，如图 D-5 所示，客户端 A 又尝试对同一资源进行条件 PUT 操作。

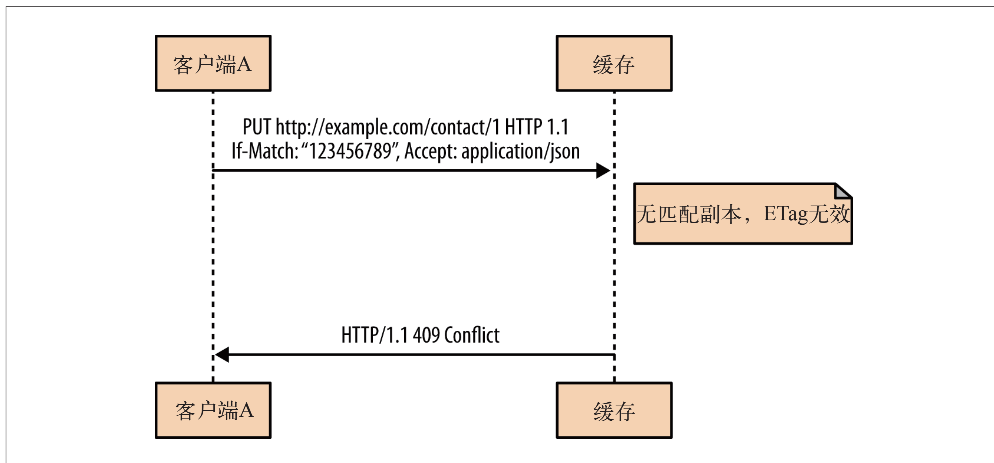


图 D-5：客户端 A 发出条件 PUT 请求

(1) 缓存收到请求并检查缓存内容，没有找到所请求的 ETag，因为这个 ETag 之前已经更新。

(2) 缓存向客户端返回一个 409 Conflict，告知客户端这个 ETag 已经失效。

附录E

身份验证工作流

在客户端到源服务器的工作流中，客户端需要向源服务器验证自己的身份（参见图 E-1）。

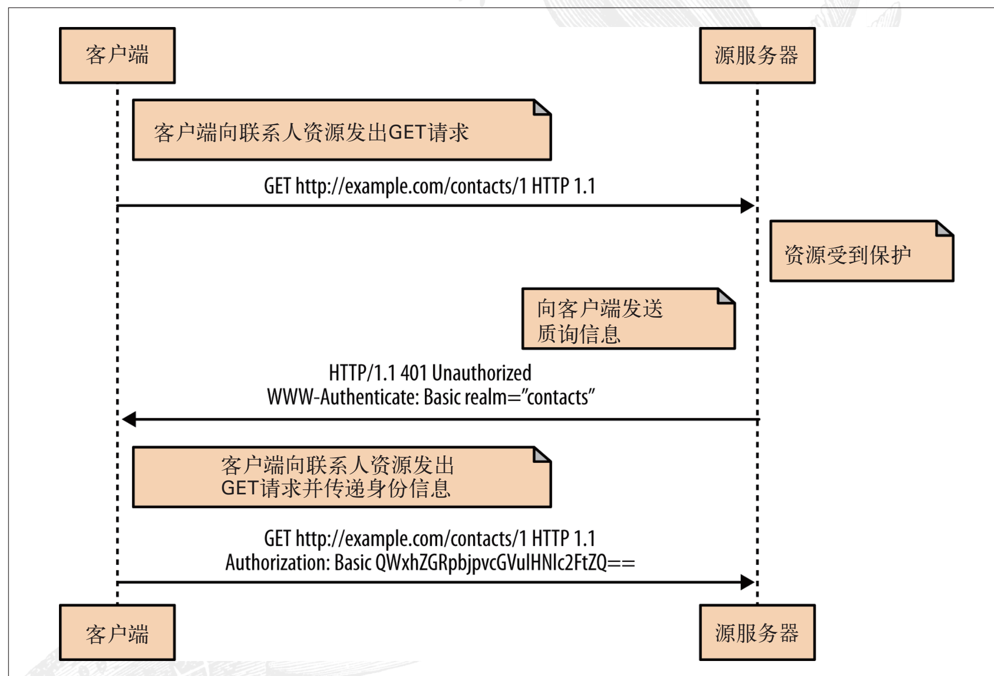


图 E-1：客户端向源服务器验证自己的身份

客户端尝试访问源服务器上的一个受保护资源。因为资源是受保护的，服务器会通过 `401`

Unauthorized 响应，向客户端发回一个质询。这个响应带有一个 WWW-Authenticate 标头（参见表 B-3），其中包含了一个或多个质询，客户端必须对这些质询做出响应，才能访问受保护资源。

客户端随后向资源发回请求，请求中提供了带有所需身份信息的 Authorization 标头。

在客户端到代理的工作流中，客户端通过安全代理访问资源，客户端必须向这个安全代理验证自己的身份。图 E-2 展示了这一过程。

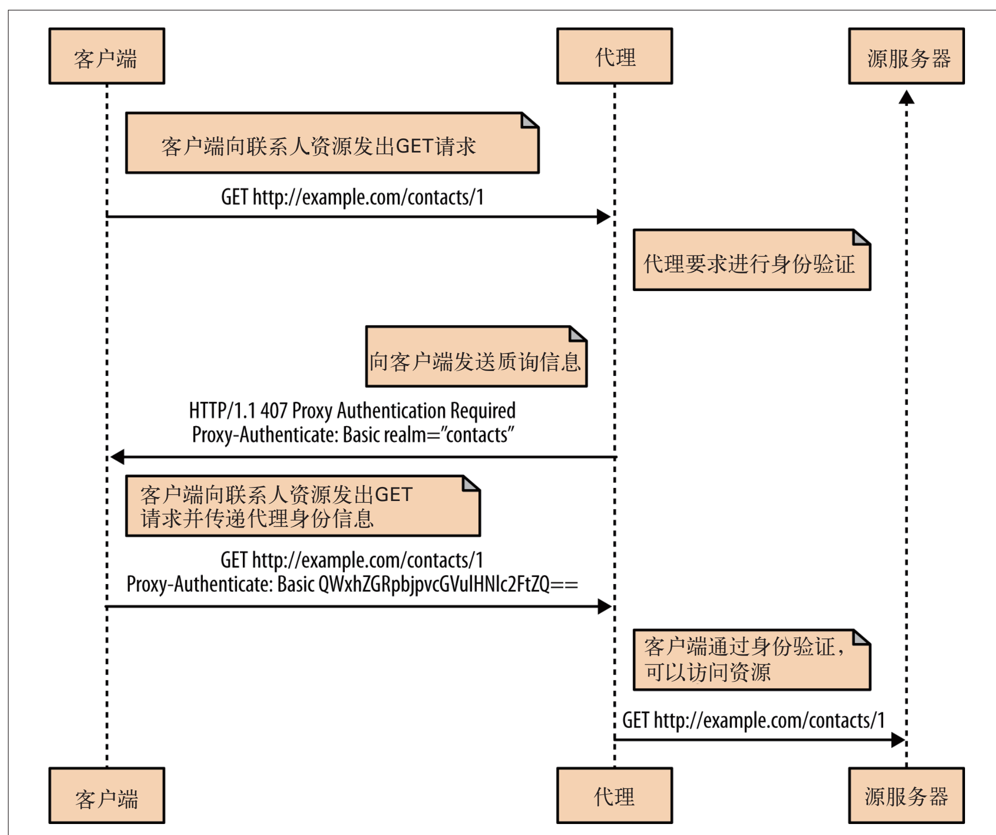


图 E-2：客户端向代理验证自己的身份

客户端尝试通过一个需要身份验证的代理，访问一个受保护资源。代理收到请求后，通过 407 Proxy Authentication Required 响应，向客户端发送质询。这个响应包含一个 Proxy-Authenticate 标头（参见表 B-3），其中包含了访问这个代理自身的一个或多个质询，客户端随后发回请求，请求中提供了带有所需身份信息的 Proxy-Authorization 标头。在通过代理的身份验证之后，如果用户尝试访问的资源是受保护的，那么还会进行源服务器身份验证。图 E-3 展示了这一过程，代理身份验证结束之后，源服务器发回质询响应。

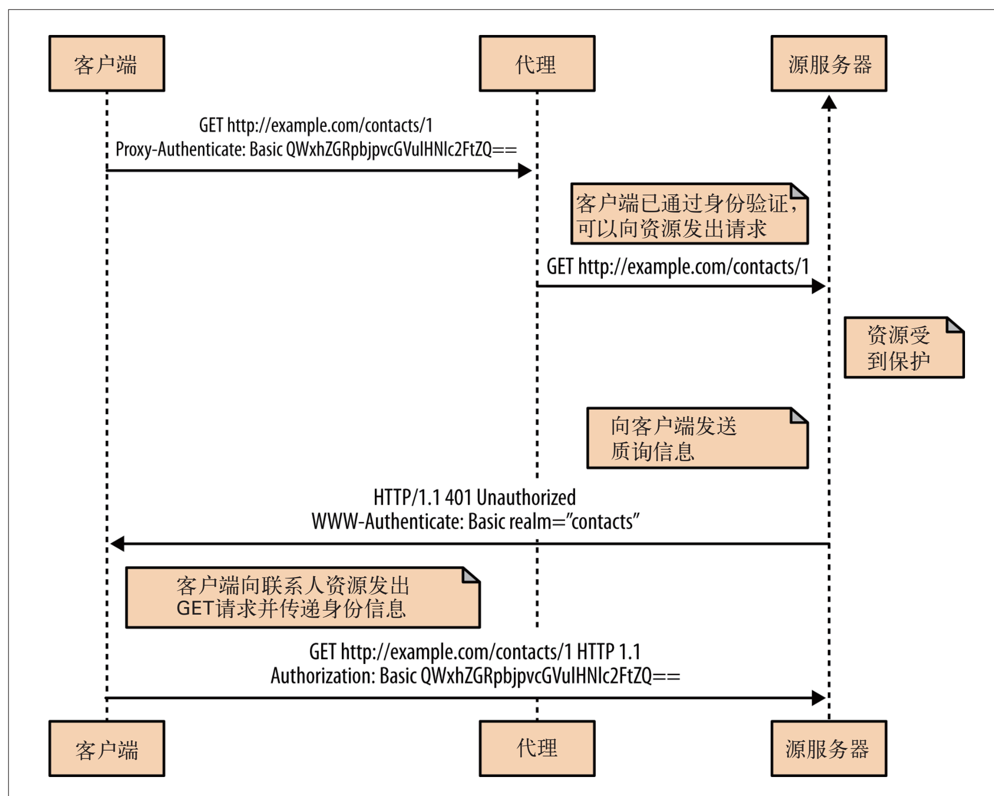


图 E-3：客户端向源服务器验证自己的身份

附录F

application/issue+json媒体类型规范

在很多复杂工程项目的生命周期，以及这些产品随后的维护中，我们需要跟踪项目相关问题的发现和解决过程。这个媒体类型规范描述了一个交互性门槛很低的文档格式。当前的规范是一个最简的定义，预期未来会加入更多的功能。

标记约定

这个文档中的关键字（MUST、MUST NOT、REQUIRED、SHALL、SHALL NOT、SHOULD、SHOULD NOT、RECOMMENDED、MAY 和 OPTIONAL）定义如 RFC 2119（参见 http://tools.ietf.org/html/rfc2119）。

问题文档

示例 F-1 所示的问题文档使用 RFC 4627 中描述的格式，文档的媒体类型为 applcation/issue+json。

示例 F-1：最小的问题文档

```
{
    "title" : "This is a very simple issue"
}
```

问题文档可以包含表 F-1 中列出的属性。

表F-1：属性语义

属性名	描　　述
id	问题的唯一数字标识符
title	问题的简短文字概述（必须具备）
description	问题的详细描述
status	问题状态的文本表示，值为：open 或 closed

issue+json 也使用链接支持超媒体，链接遵循 RFC 5988（参见 http://tools.ietf.org/html/rfc5988）中描述的链接语义。链接由一个名为 links 的数组中的一组对象定义。

安全性考虑

issue+json 具有一些所有 JSON 内容类型都有的安全问题。要获得更多的信息，请参考 RFC 4627 的第 6 节（参见 http://tools.ietf.org/html/rfc4627#section-6）。issue+json 不提供可执行内容。issue+json 文档中包含的信息不要求隐私服务或完整性服务。

交互性考虑

无法识别的文档内容应该忽略，文档不应该因为有无法识别的内容而被视为无效。

IANA考虑

这个规范定义了一个新的互联网媒体类型（RFC 6838，http://tools.ietf.org/html/rfc6838）：

```
Type name: application
Subtype name: issue+json
Required parameters: None
Optional parameters: None; unrecognised parameters should be ignored
Encoding considerations: Same as [RFC4627]
Security considerations: see [this document]
Interoperability considerations: None.
Published specification: [this document]
Applications that use this media type: HTTP
Additional information:
  Magic number(s): n/a
  File extension(s): n/a
  Macintosh file type code(s): n/a
Person & email address to contact for further information:
    Darrel Miller <darrel@tavis.ca>
Intended usage: COMMON
Restrictions on usage: None.
Author: Darrel Miller <darrel@tavis.ca>
Change controller: IESG
```

附录G

公钥加密和证书

1976 年，Whitfield Diffie 和 Martin Hellman 提出*公钥加密*，为大规模的安全通信系统带来了一大突破。Diffie 和 Hellman 提出的主要想法是，由每个实体生成和使用一个或多个键对，每个键对由一个私钥和一个公钥组成。私钥必须保密，永远不需要发送给其他通信方。而公钥可以公开分发，没有任何保密要求。这些分发的公钥随后可以由第三方使用，执行如下操作：

- 发送加密消息，这些消息只能由私钥持有者解密；
- 验证签名，只能由私钥持有者产生。

公钥加密也称为*非对称加密*（asymmetric cryptography），因为其机制使用了两个密钥，其保密要求不同，而且用途不同：

- 私钥必须保密，用于解密消息，或者产生数字签名；
- 公钥可以公开分发，没有任何保密要求，用于加密消息或验证签名。

公钥加密与*经典加密*不同。经典加密也称为*对称加密*，使用同一个密钥进行所有的操作（例如：加密和解密），这个密钥必须保密。目前所知的非对称机制，性能都不如相应的对称机制，因此人们经常使用混合技术。例如，在 TLS 中，握手协议使用非对称机制，建立一组保密的对称*会话*密钥，然后记录协议使用这组密钥，使用对称机制，保护大部分的交换消息。

但是，公钥加密带来了一个新问题：*公钥身份验证*。尽管公钥可以公开分发，但是接收方必须具有某些安全方法，得知这些密钥属于谁（即：谁是相关私钥的持有者）。如果不能正确对公钥进行身份验证，公钥接收方就容易遭到 MITM 攻击。在中间人攻击中，攻击者

将一个实体公钥替换为自己的密钥。攻击者拥有和所使用公钥相关的私钥，从而可以解密发送给这个实体的所有消息。

对公钥进行身份验证的一个常用方法是使用公钥证书。公钥证书是将一个公钥绑定到一个主体的声明，由 CA（证书授权机构）颁发和签名。这些 CA 是第三方机构，在其之上有一组实体承认其权威性和验证这种绑定的能力。也就是说，CA 检查与一个公钥相关的密钥的持有者是否也为一个名字的所有者（例如：DNS 名）。

为了介绍得更为具体，让我们来看一个例子：一个客户端执行一个 HTTP 请求，访问位于 https://webapi-book.blob.core.windows.net/ 的资源。这个资源 URI 使用 `https` 方案，因此 HTTP 请求消息必须通过受 TLS 或 SSL 保护的连接发送。安全连接通过握手协议建立。客户端首先向服务器发送一个 `client hello` 消息，其中包含客户端支持的加密机制。服务器做出响应，返回一个 `server hello` 消息，其中包含选中的加密机制，并返回一个包含服务器证书的 `certificate` 消息。证书详情请见图 G-1。

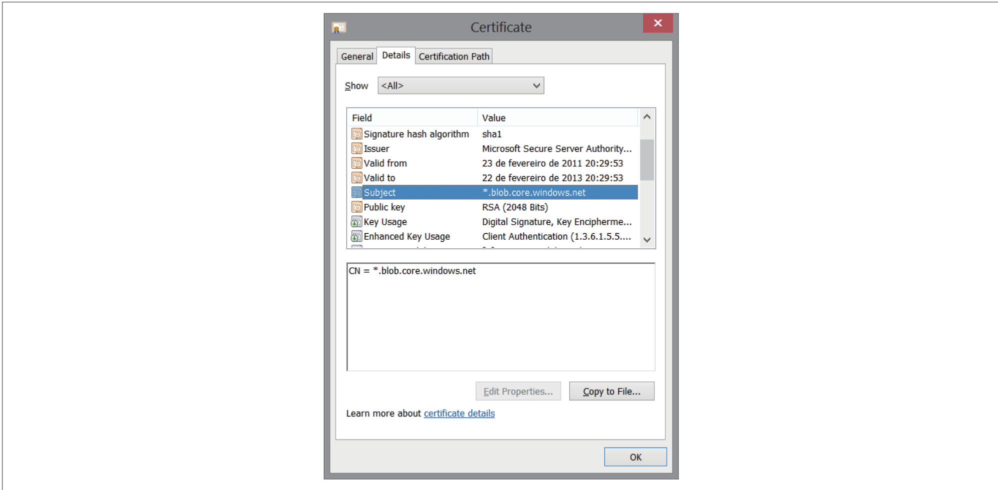


图 G-1：*.blob.core.windows.net 证书

这个证书遵循 X.509 规范（参见 http://www.itu.int/rec/T-REC-X.509），由几个字段组成，比如下面几个。

- *公钥*（public key）字段，其中包含服务器的公钥。
- *主题*（subject）字段，其中包含服务器名。在我们的示例中，subject 字段值是带有通配符的字符串 `*.blob.core.windows.net`。
- *颁发者*（issuer）字段，包含证书颁发实体的名字（CA 名）。在我们的示例中，`issuer` 字段值为 Microsoft Secure Server Authority。

这个证书还包含其颁发者产生的一个签名，因此可以通过非安全渠道存储和分发。

接收到证书之后，客户端可以使用证书中的公钥，加密一个保密的随机种子值，将其发送给服务器。这个随机种子是一个保密的值，客户端和服务器都将使用这个值，派生出保护交换字节流的会话密钥。但是，客户端首先必须确保（同时还要满足其他条件）：

- 证书的主题与 `https`URI 的主机名相符；
- 证书在从服务器到客户端传输的过程中没有遭到篡改（握手协议使用的连接是不受保护的）；
- 颁发证书的 CA（在之前的示例中，即 Microsoft Secure Server Authority）受到信任，在这种情况下有能力执行公钥和实体名之间的绑定。

最后，客户端通常会将证书的颁发者字段与一个*信任库*（trust store，其中包含受信任的颁发者名及其公钥）进行比较，完成验证。第二个任务是，使用证书颁发者的公钥（也保存在信任库中）验证证书的签名。

信任库（trust store）通常由自颁发证书组成，每个证书包含一个受到信任的 CA 名及其公钥。这些自颁发的证书由 CA 创建，通过受到认证的机制，离线分发。将一个自颁发的证书添加到这个信任库，意味着使用证书的实体：

- 决定信任这个自颁发证书的主题字段标识的 CA——也就是说，认为这个 CA 颁发的每个证书都包含一个真实的公钥绑定；
- 已经验证了这个自颁发证书所含的公钥确实属于这个 CA。

我们特别强调最后一个要求，因为一个自签名的证书不足以将一个公钥绑定到一个名字，这个验证必须通过别的途径完成。

图 G-2 展示了我们刚才所描述的模型的一个示例。

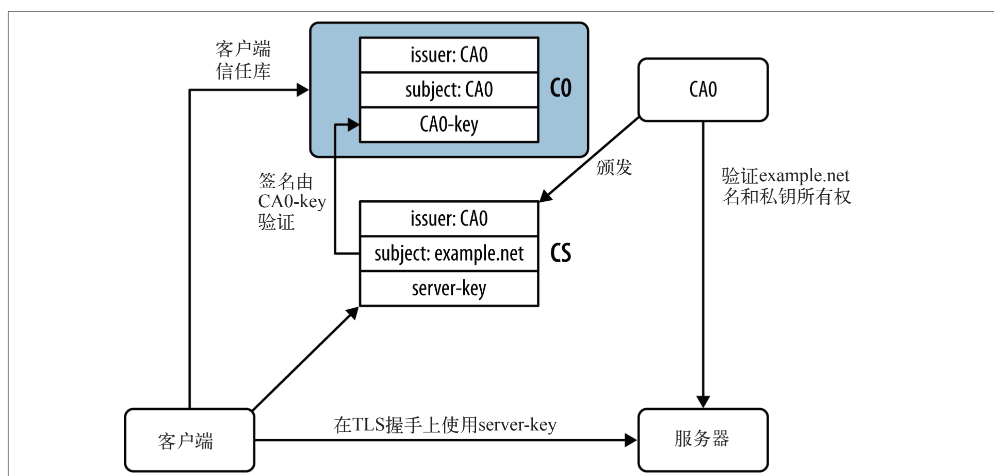


图 G-2：直接由一个受信任 CA 颁发证书

在图 G-2 中发生的事情如下。

- CA0 验证了，运行服务器的实体是与 `server-key` 相关的私钥的持有者，并且是域名 `example.net` 的合法所有者，然后颁发 CS 证书。
- 客户端信任 CA0 的认证能力，因此在检查了证书信息（也就是其公钥）的正确性之后，将 CA0 的自颁发证书安装到自己的信任库中。
- 在 TLS 握手协议中，客户端收到 CS 证书，通过检查 (1) 证书由一个受信任的 CA 颁发，(2) 通过 CA 公钥成功验证证书签名，对证书进行验证。然后，客户端使用 `server-key` 加密一个保密种子，（据 CA0 称）这个加密信息只能由 example.net 解密。

CA 颁发证书的操作，应该在对认证信息进行安全验证之后进行，特别是对公钥到名字绑定的验证。通常，CA 在一个称为*认证实践声明*（Certification Practice Statements，http://cybertrust.omniroot.com/repository/）的文档中描述这一安全验证过程。根据名字范围（DNS 名、电子邮件和居民身份号码）的不同，认证代价可能很高（例如，验证官方记录，确保一个实体是一个注册名的所有者），或者难以执行（CA 与命名机构没有任何关系）。因此，CA 可以将认证功能委托给其他 CA，这些 CA 称为*中间*（intermediate）CA 或*下属*（subordinate）CA。这种委托通过颁发一个证书完成。这个证书的颁发者是直接受到信任的 CA，主题是中间 CA。这个证书有两个作用，除了将中间 CA 名绑定到其公钥，还声明了这个颁发者的一部分认证功能委托给这个中间 CA。

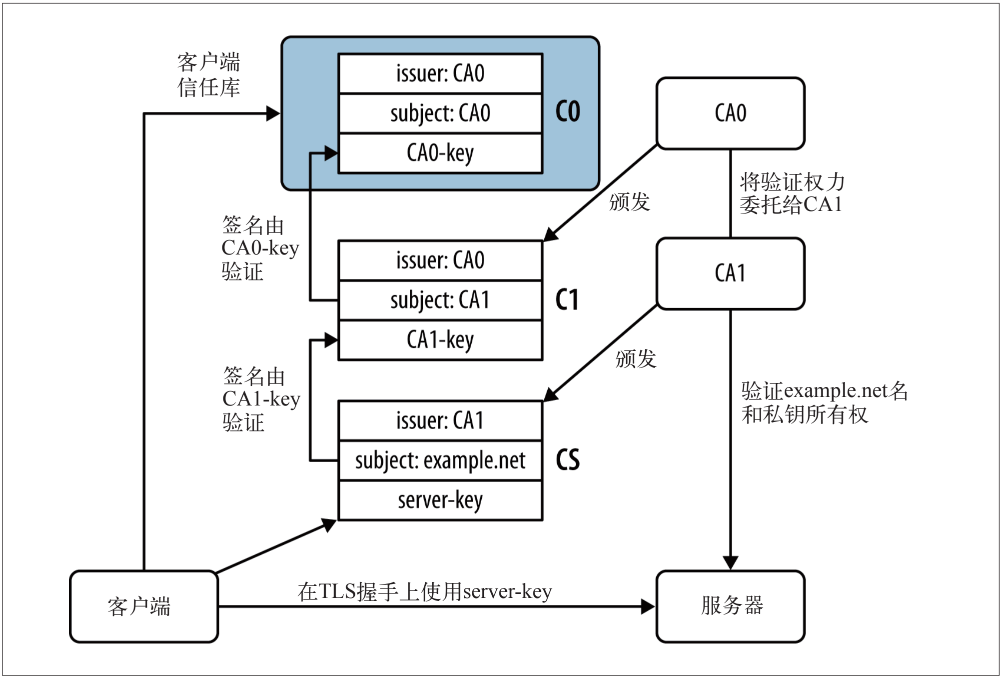


图 G-3：中间认证机构和认证路径

图 G-3 展示了这一扩展模型。

- CA0 直接受到客户端信任，通常称为根（root）CA。CA0 通过颁发中间证书 C1，将其认证功能委托给 CA1。
- CA1，而非 CA0，验证了运行服务器的实体是与 `server-key` 相关的私钥的持有者，并且是域名 `example.net` 的合法所有者。

在这个模型中，服务器证书验证要求构建一个*证书链*（certificate chain）——由从直接受到信任的 C0 证书（位于信任库中）——到服务器证书的所有证书组成，途经中间证书 C1。

再回到我们的具体场景，图 G-4 展示了 *.blob.core.windows.net 认证路径，这个路径由两个中间 CA 组成。只有根 CA（GTE Cyber Trust Global Root）受到客户端的直接信任。但是，根 CA 将自己的认证权力委托给了 Microsoft Internet Authority，Microsoft Internet Authority 又将其委托给 Microsoft Secure Server Authority。颁发服务器证书的是最后一个 CA。

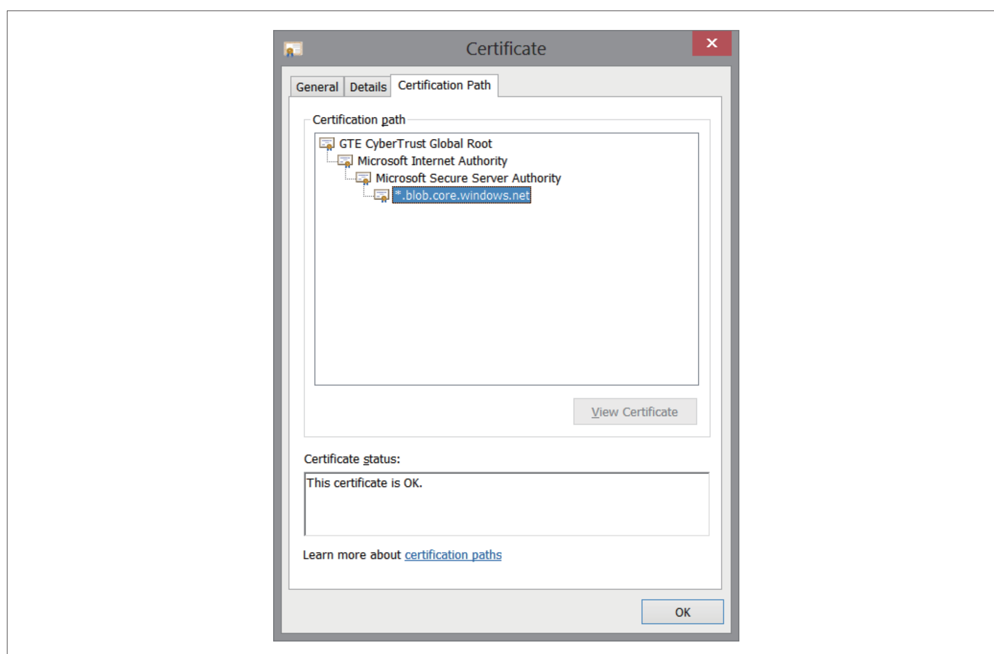


图 G-4：服务器证书路径

在 Windows 系统中，证书通过库进行管理。图 G-5 展示了一个 Windows 证书库。这些证书库按位置（*当前用户*，*本地计算机*）进行组织，具有特殊的语义。

- *私有*（Personal）库：也保存了与库中证书公钥相关的密钥。
- *受信任根证书颁发机构*（Trusted Root Certification Authorities）库：包含可以用作认证路径根的*受信任证书*。

- 中间证书颁发机构（Intermediate Certification Authorities）库：包含不是直接受到信任，但可以用来构建认证路径的 CA 证书。

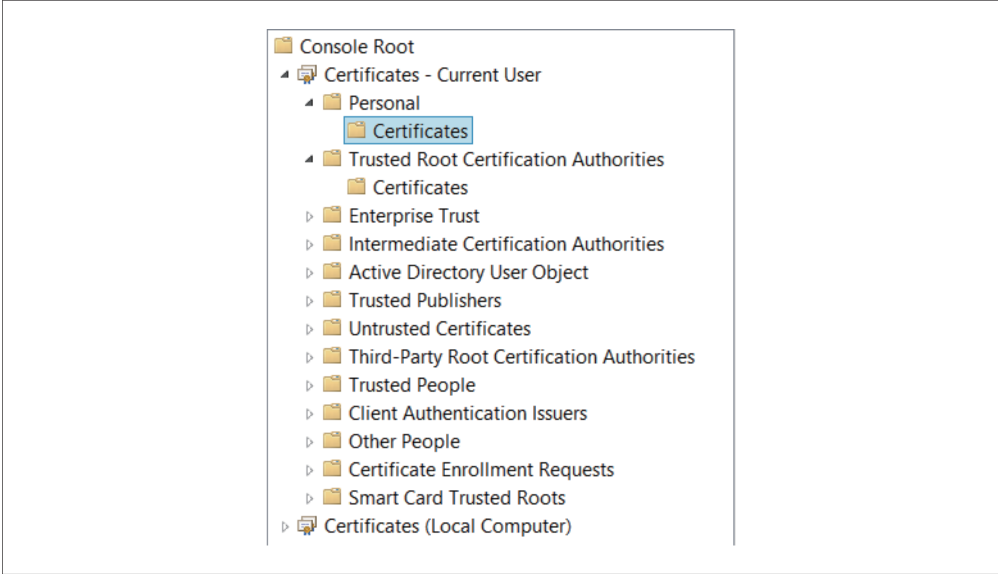


图 G-5：Windows 证书库

当 Windows 证书管理系统需要验证一个证书时，只有当认证路径的根位于受信任根证书颁发机构库中时，系统才认为这个认证路径是有效的。在向受信任根证书颁发机构库添加证书时，你必须格外谨慎，因为这个库的内容定义了谁有权颁发有效证书。例如，如果 Fiddler 工具在这个信任库中加入了一个证书，那么就可以为任何服务器动态颁发有效的证书，还可以伪装成远程服务器，拦截 HTTPS 消息。这种做法其实就是 MITM 攻击，只不过在这里用于开发和调试，用途是善意的。

撤销

证书信息的有效性可能随着时间发生改变。换句话说，如果一个攻击者获得了一个实体的私钥，那么相关的公钥就不应该再被使用。

为此，你可以使用一个包含无效（撤销）证书的 CRL（Certificate Revocation List，证书撤销清单）。例如，在之前的示例中，*.blob.core.windows.net 证书包含一个 CRL 分发点（CLR distribution point）字段，字段值为一个 CRL URI（http://mscrl.microsoft.com/pki/mscorp/crl/Microsoft%20Secure%20Server%20Authority(8).crl）。当进行证书验证时，这个 URI 可以用于获取证书撤销清单，检查证书是否已经撤销。

另一个方法是使用 OCSP（Online Certificate Status Protocol，在线证书状态协议），由客户

端直接向 CA 询问证书的当前状态，以检查证书当前是否有效。证书的验证过程应该包含这些撤销检查中的一种。

如果需要了解 X.509 证书在互联网中使用的更多信息，包括认证路径的构建和验证细节，我们推荐你阅读 PKIX IETF 工作组提出的一组规范，即 RFC 5280。

创建测试密钥和证书

在开发使用 TLS 协议的客户端和服务器时，如果能有一组用于测试的密钥和证书，对我们会很有帮助。在随后的段落中，我们将介绍如何使用一组 Windows 命令行工具，创建测试密钥和证书。但是，在开始之前，我们应该强调一点：这些密钥和证书只用于测试用途，请不要在生产环境中使用。

第一步，使用 `makecert` 工具，创建一个根证书颁发机构。[1]

```
makecert -r -n“CN=Demo Certification Authority;O=Web API Book”^
 -sv webapibook-ca.pvk ^
 -len 2048 -e 01/01/2020 -cy authority webapibook-ca.cer
```

其中的 `-r` 选项告诉 `makecert` 生成一个自签名的证书——也就是说，使用与证书中所含公钥相关的密钥进行签名的证书。这个证书将用作认证路径的根。

这个证书颁发机构的 X.500 名为 `CN=Demo Certification Authority;O=Web API Book`，其中 `CN` 和 `O` 都是名字属性：`CN` 代表*常用名*（common name），`O` 代表*组织*（organization）。私钥将使用由给定密码生成的密钥进行加密，保存在文件 webapibook-ca.pvk 中。这个私钥将用于对颁发的每个证书进行签名。

第二步，为一个虚拟的服务器 www.example.net 生成一个非对称密钥对和证书。这个任务也可以使用 `makecert` 工具完成。

```
makecert.exe -iv webapibook-ca.pvk -ic webapibook-ca.cer -n“CN=www.example.net”^
  -sv example.pvk -len 2048 -e 01/01/2020 ^
  -sky exchange example.cer -eku 1.3.6.1.5.5.7.3.1

pvk2pfx.exe -pvk example.pvk -spc example.cer -pfx example.pfx
```

这个证书由上一步生成的 CA 发布，因此 `makecert` 要同时使用这个 CA 的证书（进行命名）和 CA 私钥（进行签名）。其中的 `-eku 1.3.6.1.5.5.7.3.1` 选项给生成的证书添加了一个*增强密钥使用扩展*（enhanced key usage extension），说明这个证书可以用于 TLS 服务器身份验证。

注 1：`makecert` 是 .NET 框架工具。

.pvk 文件包含服务器的私钥，.cer 文件包含服务器的证书，证书中包含其公钥。第二步执行的最后一行使用 pvk2pfx.exe 工具，将私钥和证书都封装在一个 .pfx（个人信息交换，personal information exchange）文件中。这个 .pfx 文件使用 PKCS#12 交互性格式，进行密码资料（例如：私钥和证书）的交换，是 Windows 中进行数字证书和密钥交换最常用的格式。

最后一步，我们要为两位密码学中大名鼎鼎的虚构人物 Alice 和 Bob（参见 http://bit.ly/alice-bob）生成客户端证书。

```
makecert.exe -iv webapibook-ca.pvk -ic webapibook-ca.cer ^
  -n "CN=Alice;O=Web API book fictional characters" ^
  -sv alice.pvk -len 2048 -e 01/01/2020 -sky exchange ^
  alice.cer -eku 1.3.6.1.5.5.7.3.2
pvk2pfx.exe -pvk alice.pvk -spc alice.cer -pfx alice.pfx

makecert.exe -iv webapibook-ca.pvk -ic webapibook-ca.cer ^
  -n "CN=Bob;O=Web API book fictional characters" ^
  -pe -sv bob.pvk -len 2048 -e 01/01/2020 -sky exchange ^
  bob.cer -eku 1.3.6.1.5.5.7.3.2
pvk2pfx.exe -pvk bob.pvk -spc bob.cer -pfx bob.pfx
```

这个过程与生成服务器证书和密钥的过程相似，但有一个区别：生成的证书带有 1.3.6.1.5.5.7.3.2 扩展，说明可以用于 TLS 客户端一方的身份验证。

完成这个过程，我们应该生成了两类文件。文件 webapibook-ca.cer 包含 CA 证书，而信任这个实体进行认证的各方都应该使用这个文件。在 Windows 中，用户做出这个信任决定，会将这个证书加入到用户的受信任根证书颁发机构库中。.pfx 文件包含通信各方（www.example.net、Alice 和 Bob）的证书和密钥，应该安装在各方的私有证书库中。

O'REILLY®

ASP.NET Web API设计

如何为浏览器和移动设备等多客户端设计和构建可演化Web API？本书以ASP.NET Web API框架为例，系统介绍了相关的理论和工具，让读者全面掌握设计和实现可演化Web API的技术。

本书主要面向有经验的.NET开发人员。不过，书中关于Web API基础理论和设计的内容同样适用于Java、Ruby、PHP和Node等开发者。

- **深入理解HTTP，以及API开发的概念和风格**
- **ASP.NET Web API基础知识，包括该框架处理HTTP请求的生命周期**
- **以“问题跟踪”API为例，探讨用Collection+JSON实现超媒体支持**
- **采取BDD（行为驱动开发）方式开发ASP.NET Web API，实现和改进应用**
- **探索可响应变化的客户端技术，使客户端便于使用超媒体API**
- **全面介绍ASP.NET Web API的内部工作机制，包括安全性和可测试性**

Glenn Block曾任职微软，推动了ASP.NET Web API早期版本的开发。

Pablo Cibraro拥有十余年使用微软技术设计和实现大型分布式系统的经验。

Pedro Félix是一位软件工程师和教授，研究方向为基础设施实现、用户身份管理，以及访问控制。

Howard Dierking是微软ASP.NET Web API团队的产品经理，当前的工作重点包括NuGet和Katana项目。

Darrel Miller是Tavis软件公司的创始人之一，帮助人们学习如何在业务应用中采用REST架构风格。

“这本书提供了及时而全面的指南，帮助人们使用ASP.NET Web API构建坚实的系统，融合了ASP.NET Web API团队的经验与软件业界多年的专业积累。”

——Scott Guthrie
微软“云和企业”副总裁

“第一次获悉本书时，我就急切地想读，原因有二。首先，我当前的工作中涉及为应用设计API以使之与多种系统交互。其次，是因为本书的一位作者Glenn Block，我认识他有段时间了，我见过他，与他有过对话，读了他的大量博文，我确定本书不会让人失望，而事实证明确实如此。”

——**Joseph Guadagno**
亚利桑那Southeast Valley .NET用户组创始人

封面设计：Randy Comer **张健**

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机//程序设计/.NET技术

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-37772-2



ISBN 978-7-115-37772-2
定价：99.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks